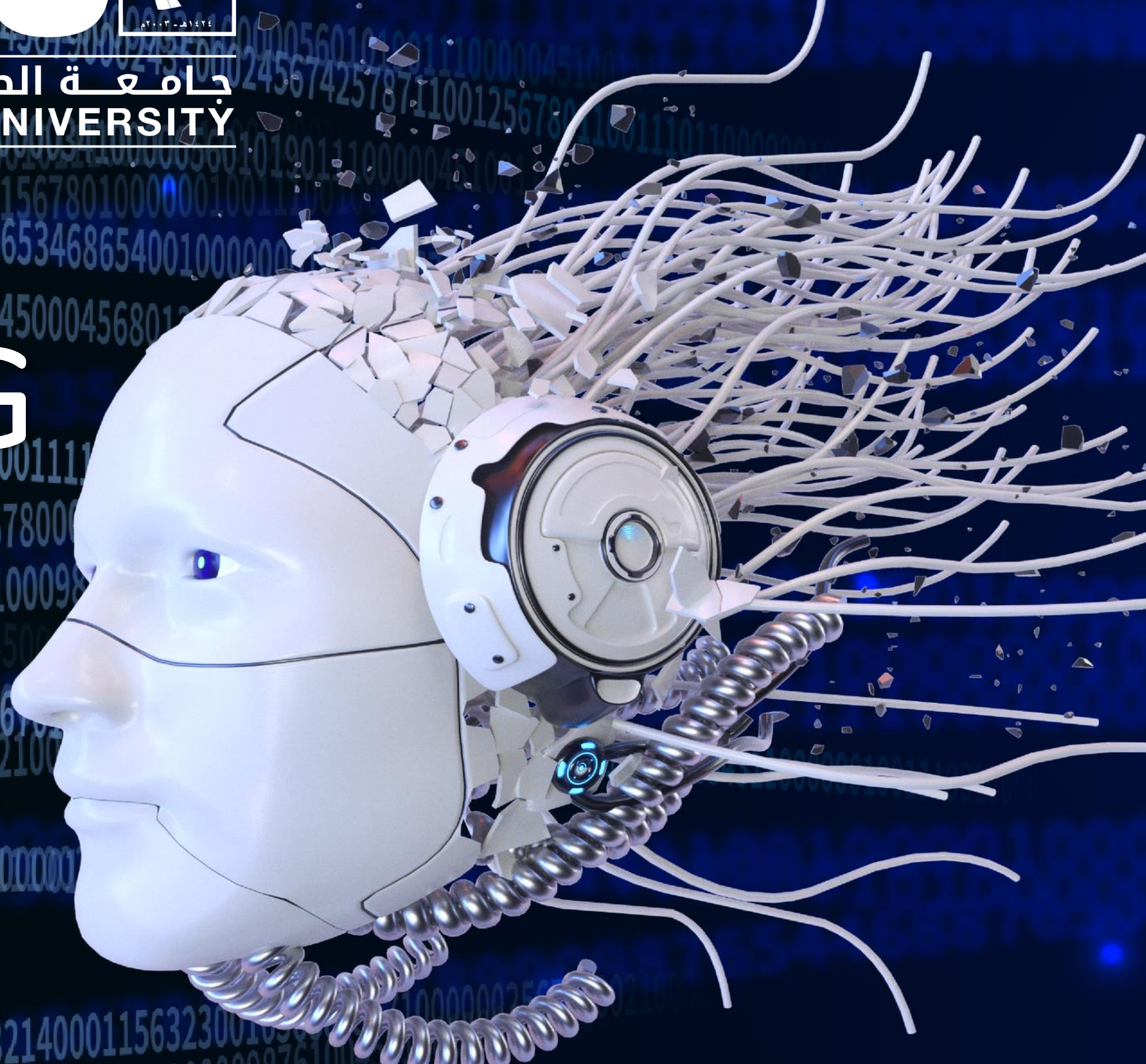
# DEEP LEARNING

Assoc. Prof. Dr. Salha Alzahrani

Lecture Notes for MSc. in Data Science



My Notes

## In this chapter, we will cover the following topics:

- Intuition and justification for convolutional neural networks (CNNs)
- Convolutional layers
- Stride and padding
- 1D, 2D, and 3D convolutions
- Backpropagation in convolutional layers
- Pooling layers
- The structure of a convolutional network
- Improving the performance of CNNs
- A CNN example with Keras and MNIST
- A CNN example with Keras and CIFAR-10

# Watch this video: What is CNN and how it works!

https://youtu.be/K_BHmztRTpA

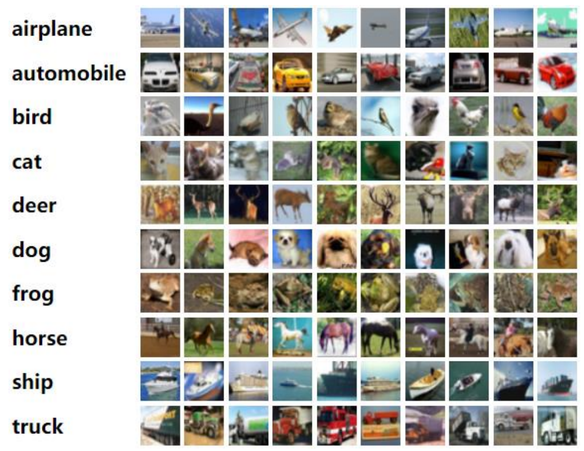
## Intuition and justification for CNN

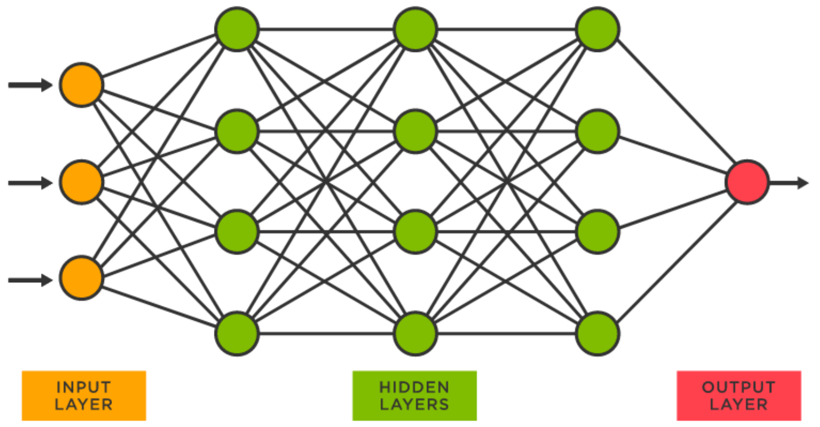
- The information we extract from sensory inputs is often determined by their context. With images, we can assume that nearby pixels are closely related, and their collective information is more relevant when taken as a unit. Conversely, we can assume that individual pixels don't convey information related to each other. For example, to recognize letters or digits, we need to analyze the dependency of pixels close by, because they determine the shape of the element. In this way, we could figure the difference between, say, a 0 or a 1. The pixels in an image are organized in a two-dimensional grid, and if the image isn't grayscale, we'll have a third dimension for the color maps.

- Alternatively, a magnetic resonance image (MRI) also uses three-dimensional space. If we wanted to feed an image to a fully-connected neural network (MLP), we had to reshape it from two-dimensional to a one-dimensional array. CNNs are built to address this problem: how to make information pertaining to neurons that are closer more relevant than information coming from neurons that are further apart. In visual problems, this translates into making neurons process information coming from pixels that are near to one another. With CNNs, we'll be able to feed one-, two-, or three-dimensional inputs and the network will produce an output of the same dimensionality.

## Intuition and justification for CNN

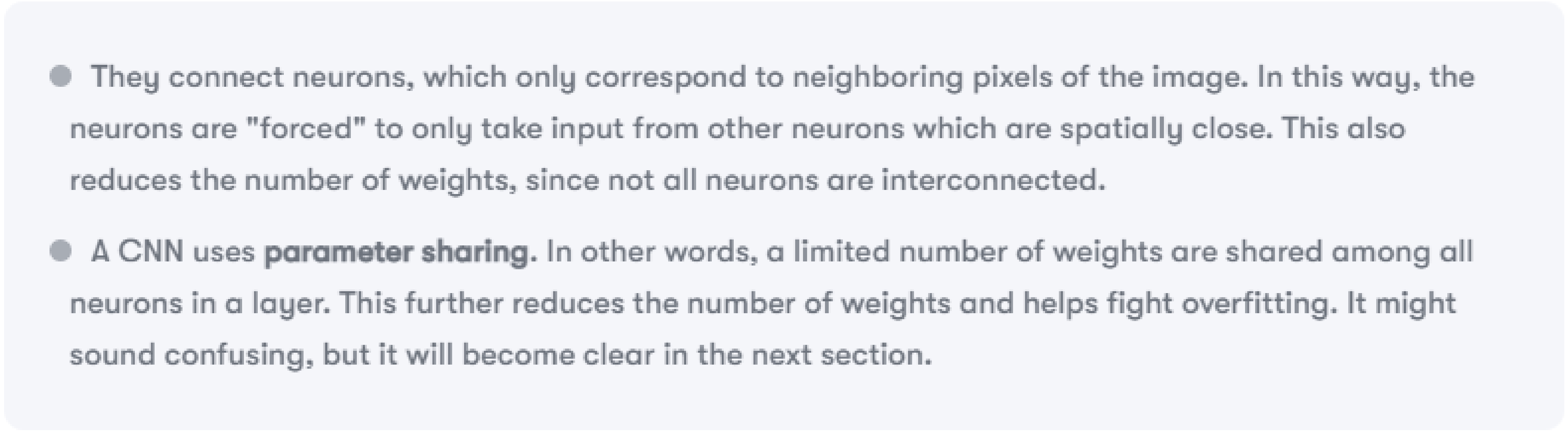
- At the end of the previous chapter, we tried to classify the **CIFAR-10 images** using network of fully-connected layers with little success because:
    1) Using fully-connected MLP to classify images leads to **overfit**. Let's analyze the first hidden layer of that network, which has 1,024 neurons. The input size of the image is 32x32x3 = 3,072. Therefore, the first hidden layer had a total of 2072 * 1024 = 314, 5728 weights which is a big number!
    2) It is also **memory inefficient**.
    3) Each input neuron (or pixel) is fully connected to every neuron in the hidden layer. Because of this, the network cannot take advantage of the spatial proximity of the pixels, since it doesn't have a way of knowing which pixels are close to each other.





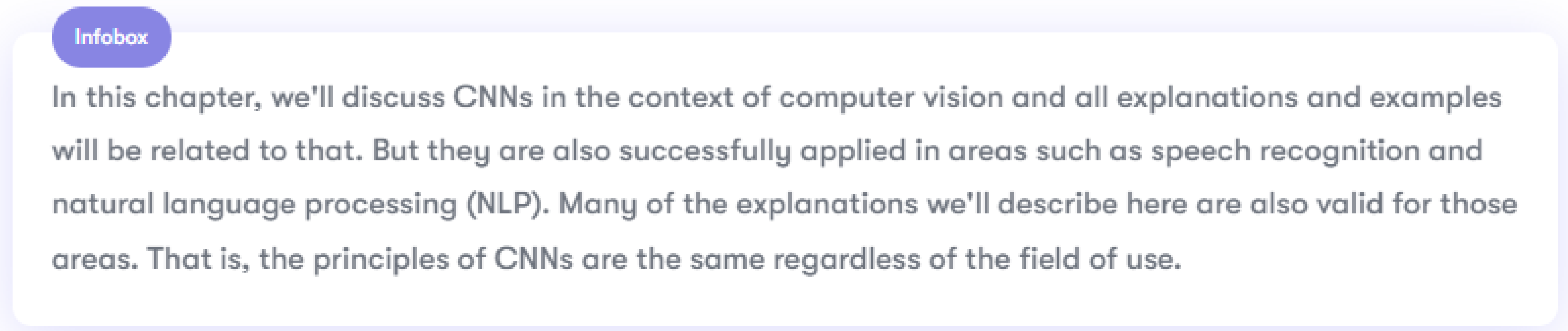
## Intuition and justification for CNN

- By contrast, CNNs have properties that provide an effective solution to these problems:



- They connect neurons, which only correspond to neighboring pixels of the image. In this way, the neurons are "forced" to only take input from other neurons which are spatially close. This also reduces the number of weights, since not all neurons are interconnected.
- A CNN uses **parameter sharing.** In other words, a limited number of weights are shared among all neurons in a layer. This further reduces the number of weights and helps fight overfitting. It might sound confusing, but it will become clear in the next section.

## Intuition and justification for CNN

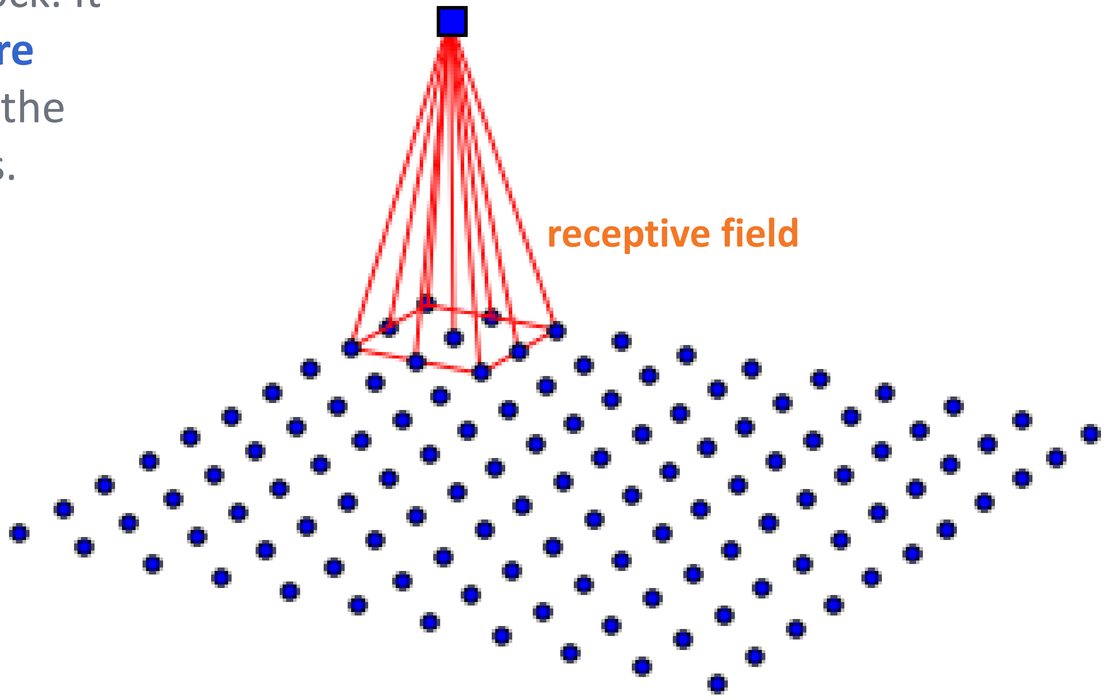


Infobox

In this chapter, we'll discuss CNNs in the context of computer vision and all explanations and examples will be related to that. But they are also successfully applied in areas such as speech recognition and natural language processing (NLP). Many of the explanations we'll describe here are also valid for those areas. That is, the principles of CNNs are the same regardless of the field of use.

## Convolutional layers

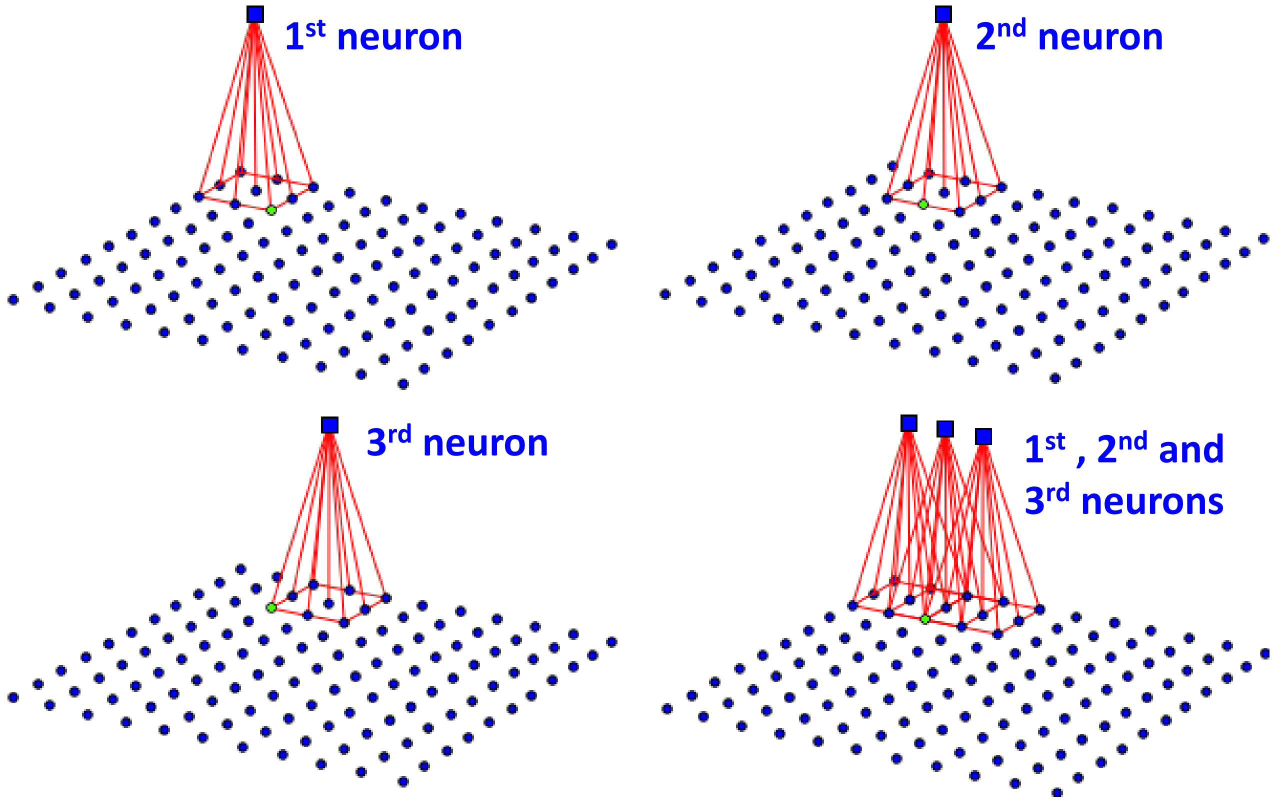
- The convolutional layer is the most important building block. It consists of a set of **filters (also known as kernels or feature detectors),** where each filter is applied across all areas of the input data. A filter is defined by a set of learnable weights.
  - Each input neuron represents the color intensity of a pixel (assume it's a grayscale image for simplicity).
  - First, apply a 3x3 filter in the top-right corner of the image. Each input neuron is associated with a single weight of the filter. It has nine weights, because of the nine input neurons.
  - Note that we can choose another size for the filter (2x2, 4x4, 5x5, and so on).
  - The output of the filter is a **weighted sum** of its inputs. Its purpose is to highlight a specific feature in the input, for example, an edge or a line. The group of nearby neurons, which participate in the input are called the **receptive field**.
  - The filter output represents the activation value of a neuron in the next layer. **The neuron will be active, if the feature is present at this spatial location.**



**A two-dimensional input layer of a neural network**

## Convolutional layers

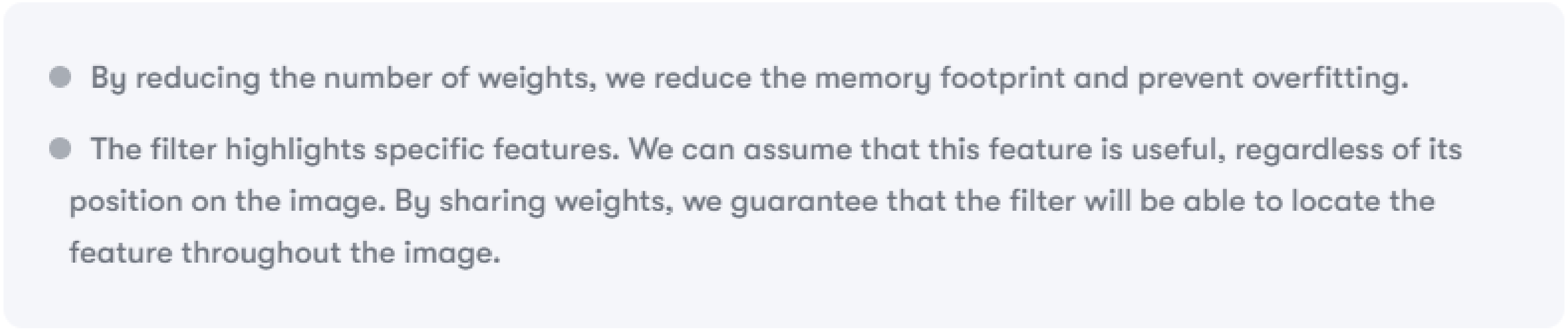
- So far, we've calculated the activation of a single neuron.
- **What about the others?** It's simple! For each new neuron, we'll **slide** the filter across the input image, and compute its output (the weighted sum) with each new set of input neurons.
- As the filter moves across the image, we compute the new activation values for the neurons in the output slice.



Computing the activations of the next two positions (one pixel to the right)

## Convolutional layers

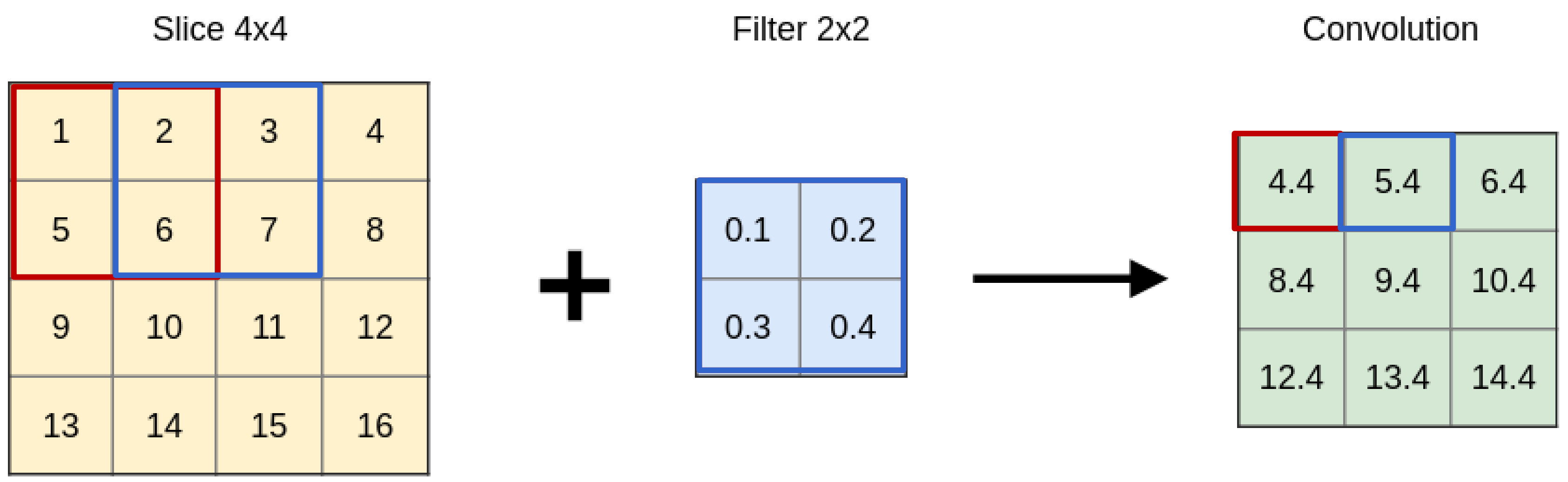
- **By saying "slide",** we mean that the weights of the filter don't change across the image. In effect, we'll use the same nine filter weights to compute the activations of all output neurons, each time with a different set of input neurons. We call this **parameter sharing**, and we do it for two reasons:



- By reducing the number of weights, we reduce the memory footprint and prevent overfitting.
- The filter highlights specific features. We can assume that this feature is useful, regardless of its position on the image. By sharing weights, we guarantee that the filter will be able to locate the feature throughout the image.

## Convolutional layers

- To compute all output activations, we'll repeat the process until we've moved across the whole input.
- The spatially arranged neurons are called **depth slices** (or a **feature map**), implying that there is more than one slice. The slice can serve as an input to other layers in the network. Finally, we can use activation function after each neuron. The most common activation function is the **ReLU**.

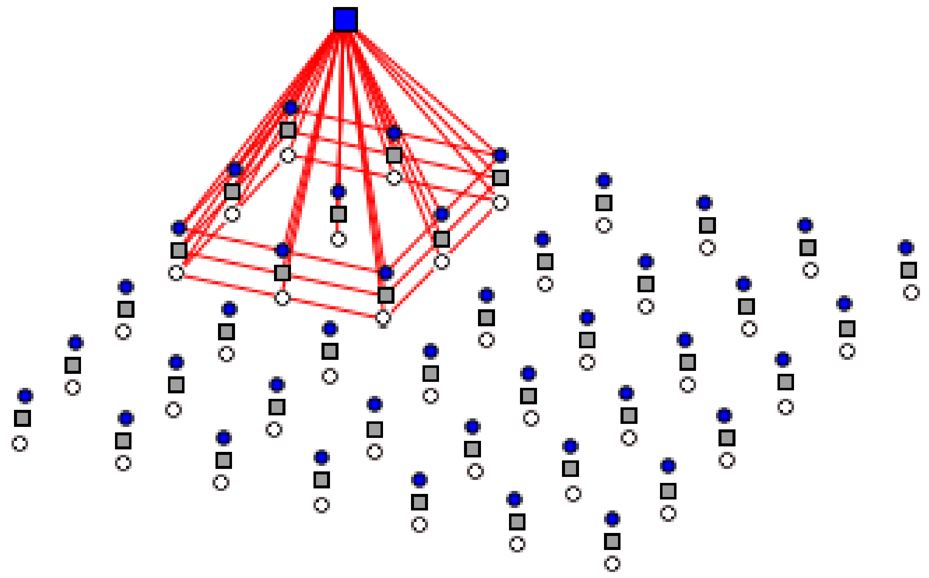


An example of convolution with a 2x2 filter over a 4x4 slice. The output is a 3x3 slice

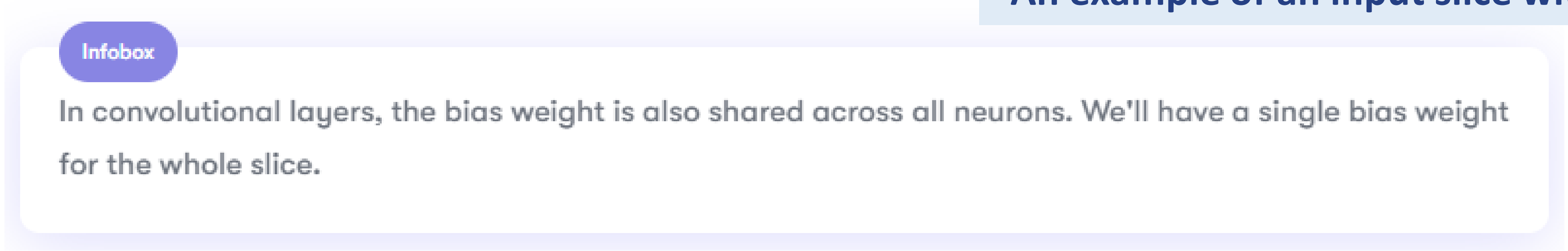
## Convolutional layers

- So far, we have described the **one-to-one slice relation**, where the output is a single slice, which takes input from another slice (or an image) which works well in grayscale.

- **But how do we adapt it for color images (n to 1 relation)?** Once again, it's simple! First, we'll split the image in color channels. In the case of RGB, that would be three. We can think of each color channel as a depth slice, where the values are the pixel intensities for the given color (R, G, or B).



An example of an input slice with depth 3

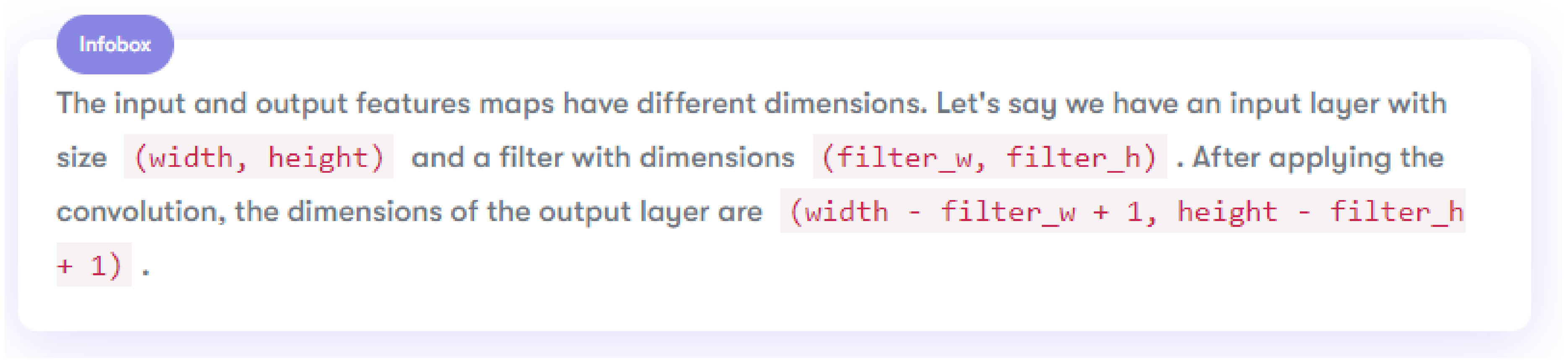


Infobox

In convolutional layers, the bias weight is also shared across all neurons. We'll have a single bias weight for the whole slice.

## Convolutional layers

- The combination of slices is called input volume with a depth of 3. A unique 3x3 filter is applied to each slice. The activation of one output neuron is just the weighted sum of the filters applied across all slices.
- In other words, we'll combine the three filters in one big 3 x 3 x 3 + 1 filter with 28 weights (we added depth and a single bias).
- Then, we'll compute the weighted sum by applying the relevant weights to each slice.

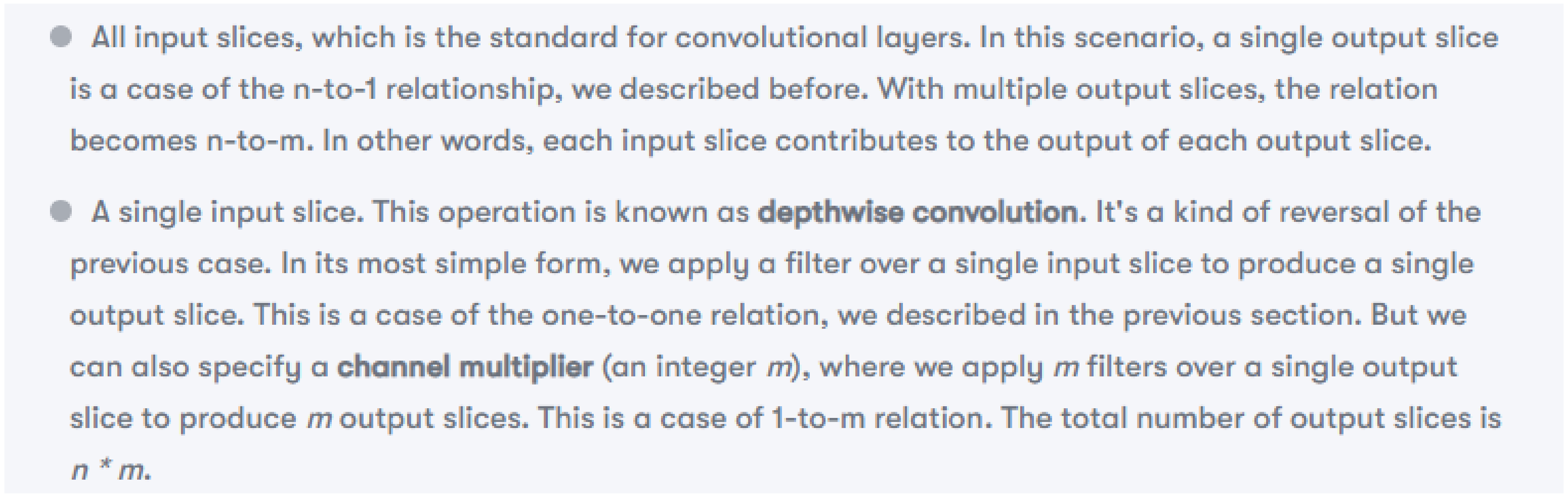


Infobox

The input and output features maps have different dimensions. Let's say we have an input layer with size `(width, height)` and a filter with dimensions `(filter_w, filter_h)`. After applying the convolution, the dimensions of the output layer are `(width - filter_w + 1, height - filter_h + 1)`.

## Convolutional layers

- As we mentioned, a filter highlights a specific feature, such as edges or lines. But, in general, many features are important and we'll be interested in all of them.
- **How do we highlight them all?** We'll apply multiple filters across the set of input slices. Each filter will generate a unique output slice, which highlights the feature, detected by the filter (n to m relation). An output slice can receive input from:



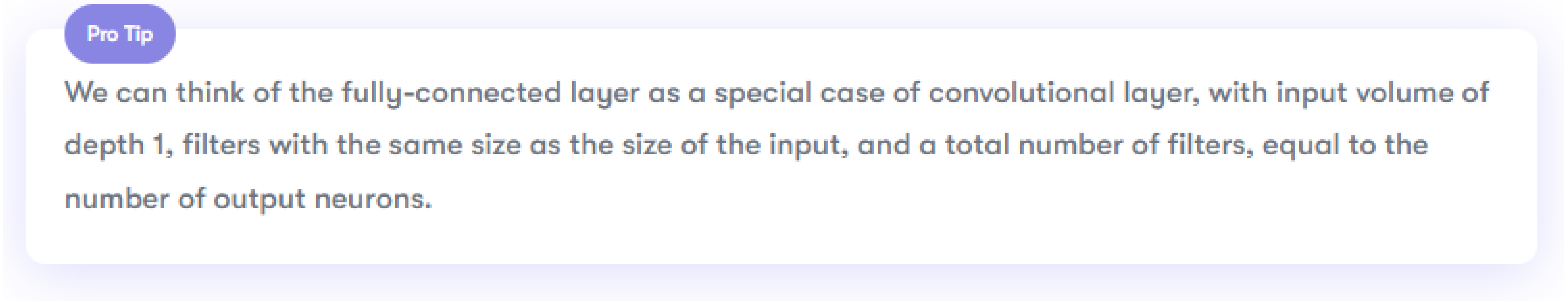
- All input slices, which is the standard for convolutional layers. In this scenario, a single output slice is a case of the n-to-1 relationship, we described before. With multiple output slices, the relation becomes n-to-m. In other words, each input slice contributes to the output of each output slice.
- A single input slice. This operation is known as **depthwise convolution**. It's a kind of reversal of the previous case. In its most simple form, we apply a filter over a single input slice to produce a single output slice. This is a case of the one-to-one relation, we described in the previous section. But we can also specify a **channel multiplier** (an integer *m*), where we apply *m* filters over a single output slice to produce *m* output slices. This is a case of 1-to-m relation. The total number of output slices is *n * m*.

## Convolutional layers

- Let's denote the width and height of the filter with $F_w$ and $F_h$, the depth of the input volume with *D*, and the depth of the output volume with *M*. Then, we can compute the total number of weights *W* in a convolutional layer with the following equation:

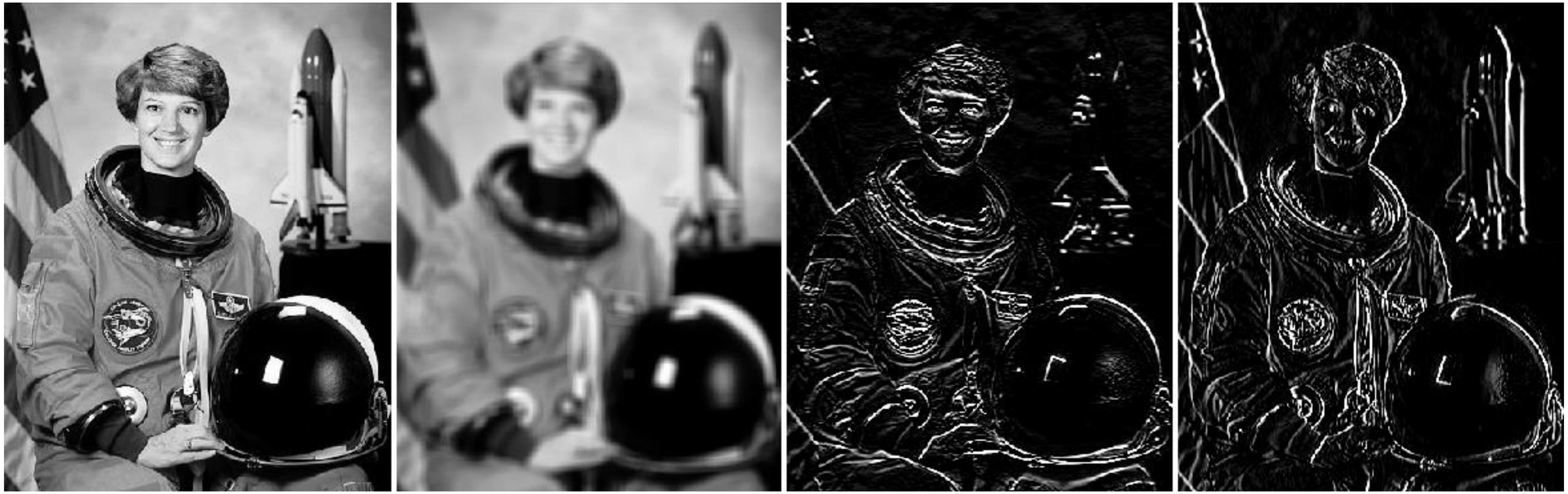
$$W = (D * F_w * F_h + 1) * M$$

- Let's say we have three slices and want to apply four 5x5 filters to them. Then, the convolutional layer will have a total of *(3x5x5 + 1) * 4 = 304* weights, and four output slices (output volume with depth of 4), one bias per slice. The filter for each output slice will have three 5x5 filter patches for each of the three input slices and one bias for a total of 3x5x5 + 1 = 76 weights. The combination of the output maps is called **output volume** with a **depth** of four.


Pro Tip

We can think of the fully-connected layer as a special case of convolutional layer, with input volume of depth 1, filters with the same size as the size of the input, and a total number of filters, equal to the number of output neurons.

## An example of convolution operation

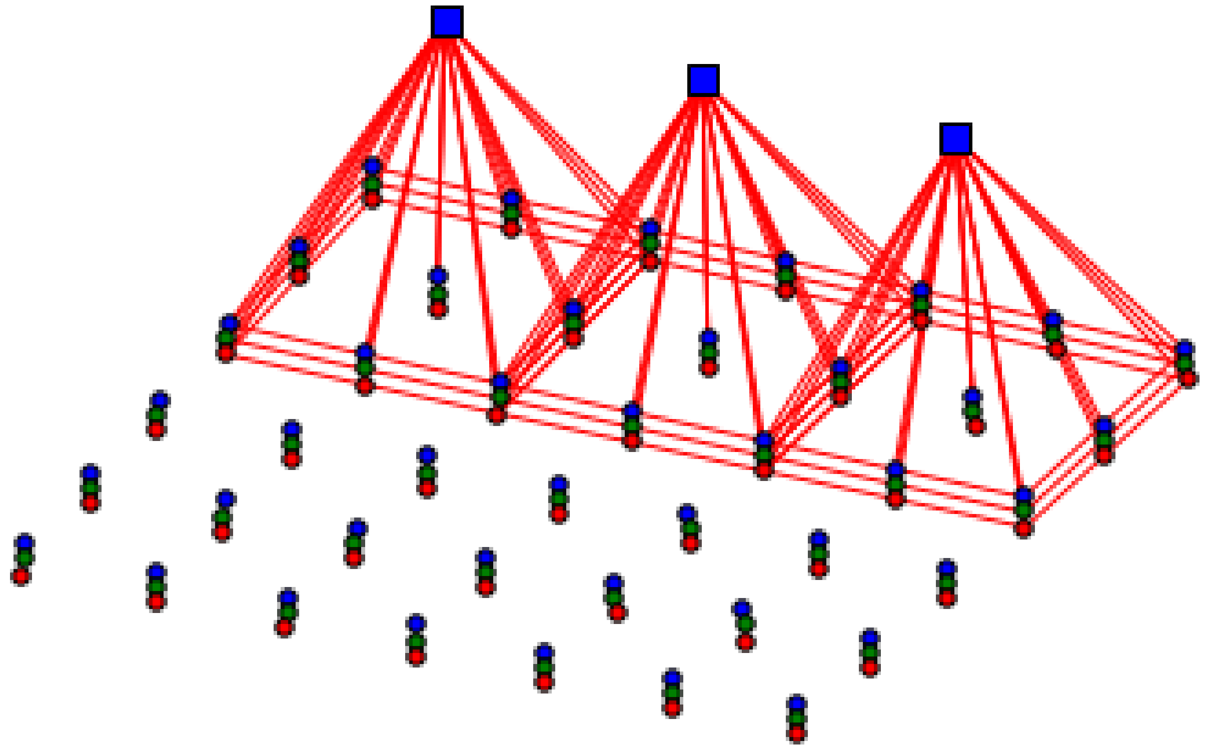


**The first image is the grayscale input. The second image is the result of a 10 x 10 blur filter. The third and fourth images use vertical Sobel edge detector**

- In this example, we used filters with hard-coded weights to visualize how the convolution operation works in neural networks. In reality, the weights of the filter will be set during the network training.
- All we'll need to do is define the network architecture, such as the number of convolutional layers, depth of the output volume, and the size of the filters. The network will figure out the features, highlighted by each filter during training.

## Stride and padding in convolutional layers

- Until now, we assumed that sliding of the filter happens one pixel at a time, but that's not always the case. We can slide the filter multiple positions. This parameter of the convolutional layers is called **stride**.
- Usually, the stride is the same across all dimensions of the input.



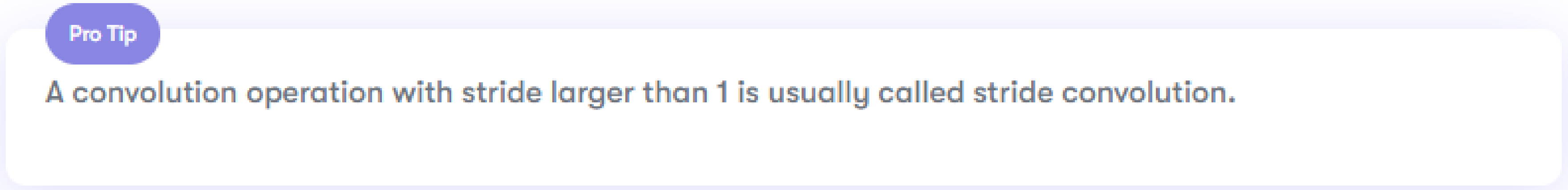
With stride 2, the filter is translated by two pixels at a time

## Stride and padding in convolutional layers

- By using a stride larger than 1, we reduce the size of the output slice. In the previous section, we introduced a simple formula for the output size, which included the sizes of the input and the kernel. Now, we'll extend it to also include the stride:

$$((\text{width} - \text{filter\_w}) / \text{stride\_w} + 1, ((\text{height} - \text{filter\_h}) / \text{stride\_h} + 1).$$

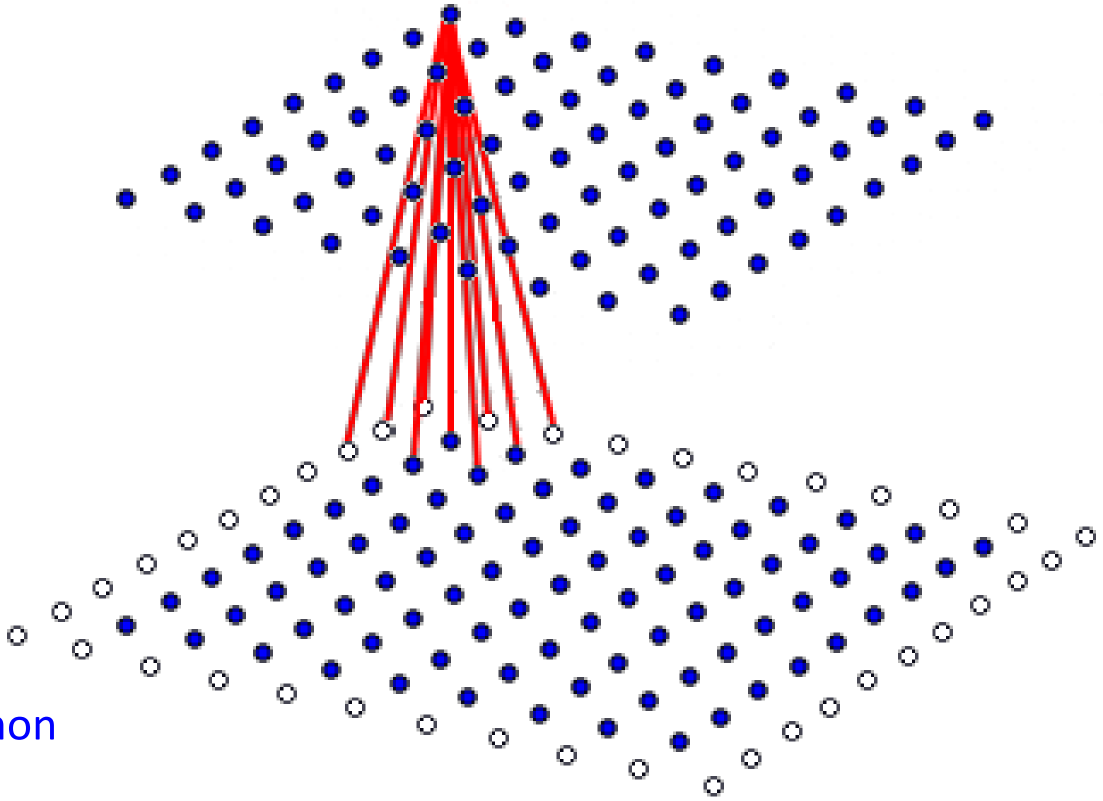
- For example, the output size of a square slice generated by a 28x28 input image, convolved with a 3x3 filter with stride 1, would be 28 - 3 + 1 = 26. But with stride 2, we get (28 - 3) / 2 + 1 = 13.
- The main effect of the larger stride is an increase in the receptive field of the output neurons. Let's explain this with an example. If we use stride 2, the size of the output slice will be roughly four times smaller than the input. In other words, one output neuron will "cover" area, which is four times larger, compared to the input neurons. The neurons in the following layers will gradually capture input from larger regions from the input image. This is important, because it would allow them to detect larger and more complex features of the input.


Pro Tip

A convolution operation with stride larger than 1 is usually called stride convolution.

## Stride and padding in convolutional layers

- The convolution operations we have discussed until now have produced smaller output than the input. But, in practice, it's often desirable to control the size of the output.
- We can solve this by **padding** the edges of the input slice with rows and columns of zeros before the convolution operation.
- The most common way to use padding is to produce output with the same dimensions as the input.
- The white neurons represent the padding.
- The input and the output slices have the same dimensions (dark neurons). This is the most common way to use padding.



Convolutional layer with padding 1

## Stride and padding in convolutional layers

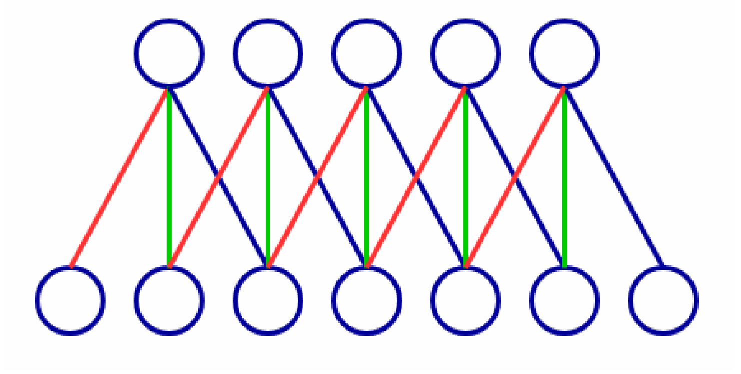
- The newly padded zeros will participate in the convolution operation with the slice, but they won't affect the result. The reason is that, even though the padded areas are connected with weights to the following layer, we'll always multiply those weights by the padded value, which is 0.
- We'll now add padding to the formula of the output size. Let the size of the input slice be *$I=(I_w, I_h)$*, the size of the filter *$F=(F_w,F_h)$*, the stride *$S=(S_w,S_h)$*, and the padding *$P=(P_w,P_h)$*.
- Then the size *$O=(O_w,O_h)$* of the output slice is given by the following equations:

$$O_w = \frac{I_w + 2P_w - F_w}{S_w} + 1$$

$$O_h = \frac{I_h + 2P_h - F_h}{S_h} + 1$$

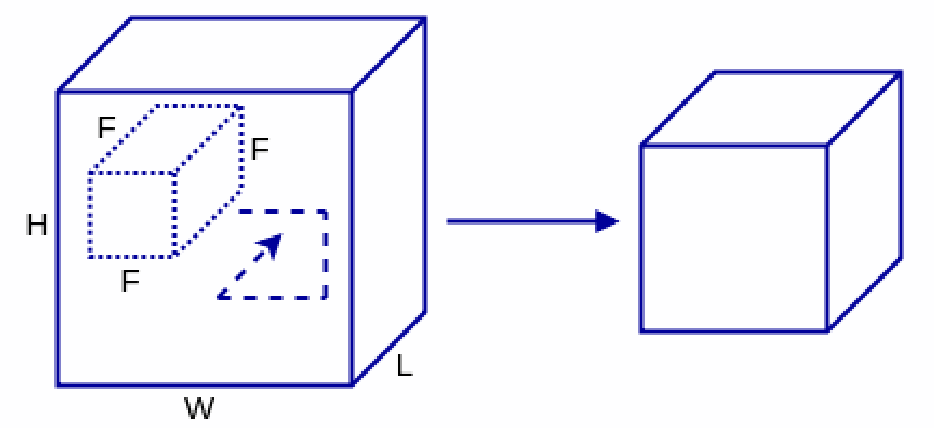
## 1D, 2D, and 3D convolutions

- Until now, we've used 2D convolutions, where the input and output neurons were arranged in a two-dimensional grid. This works very well for images. But we can also have 1D and 3D convolutions, where the neurons are arrange in one-dimensional or three-dimensional space respectively.
- In all cases, the filter has the same number of dimensions as the input and the weights are shared across the input. For example, we would use 1D convolution with time-series data, because the values are arranged across a single time axis.



**1D convolution**

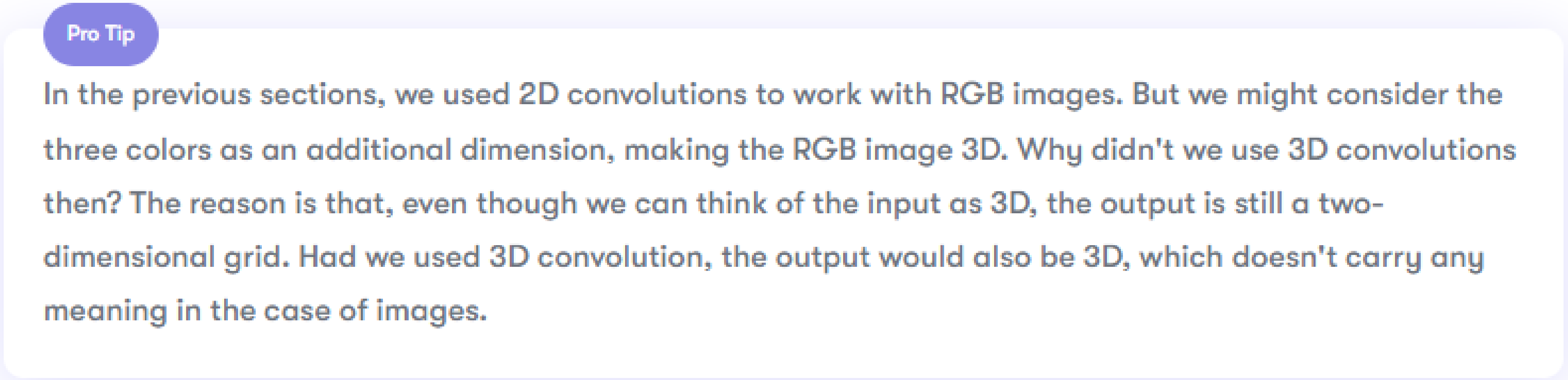
The weights with the same color (red, green, or blue) share the same value. The output of 1D convolution is also 1D.



**3D convolution**

The input has dimensions H/W/L and the filter has a single size F for all dimensions. The output is also 3D.

## 1D, 2D, and 3D convolutions


Pro Tip

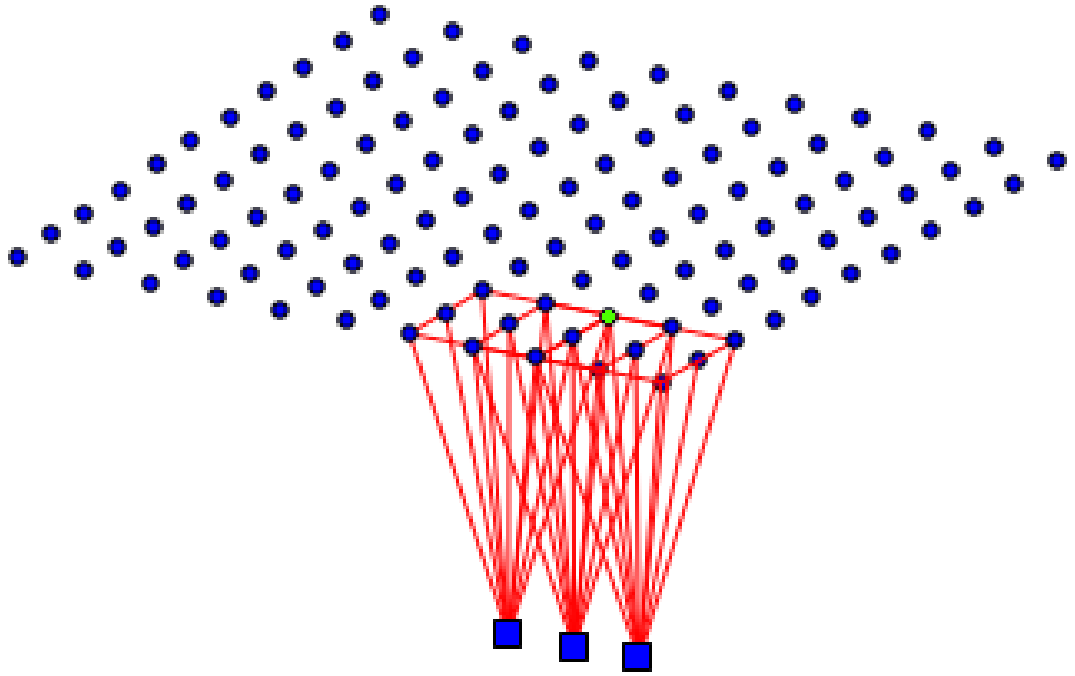
In the previous sections, we used 2D convolutions to work with RGB images. But we might consider the three colors as an additional dimension, making the RGB image 3D. Why didn't we use 3D convolutions then? The reason is that, even though we can think of the input as 3D, the output is still a two-dimensional grid. Had we used 3D convolution, the output would also be 3D, which doesn't carry any meaning in the case of images.

## 1x1 convolutions

- 1x1 (pointwise) convolution is a special case of convolution, where each dimension of the convolution filter is of size 1 (1x1 in 2D convolutions and 1x1x1 in 3D).
- At first this doesn't make sense—a 1x1 filter doesn't increase the receptive field size of the output neurons. The result of such convolution would be just pointwise scaling.
- But it can be useful in another way - we can use them to change the depth between the input and output volumes.
- To understand this, let's recall that in the general case we have input volume with a depth of *D* slices and *M* filters for *M* output slices. Each output slice is generated by applying a unique filter over all input slices. If we use a 1x1 filter and *D != M*, we'll have output slices of the same size, but with different volume depth. At the same time, we won't change the receptive field size between input and output.
- The most common use case is to reduce the output volume, or $D > M$ (dimension reduction), nicknamed the "bottleneck" layer.

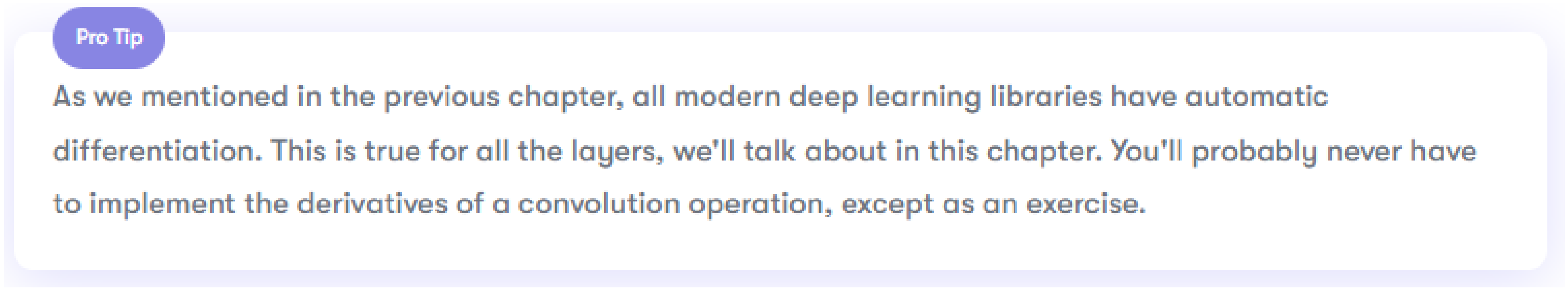
## Backpropagation in convolutional layers

- Previously, we talked about backpropagation in fully-connected layers.
- The same rule applies for convolutional layers, where the neurons are locally-connected. In the convolutional layers, we observed how a neuron participates in the inputs of several output neurons.
- This is illustrated in the following diagram, where we can se a convolution operation with 3x3 filter. The green neuron will participate in the inputs of 9 output neurons, arranged in a 3x3 pattern. Conversely, the same neurons will route the gradient in the backward pass. Thus, the backward pas of a convolution operation is another convolution operation with the same parameters, but with spatially-flipped filter. This operation is known as transposed convolution (or deconvolution). As we'll see later in the course, it has other applications besides backpropagation.



**The backward pass of a convolution operation is also a convolution**

## Backpropagation in convolutional layers



Pro Tip

As we mentioned in the previous chapter, all modern deep learning libraries have automatic differentiation. This is true for all the layers, we'll talk about in this chapter. You'll probably never have to implement the derivatives of a convolution operation, except as an exercise.

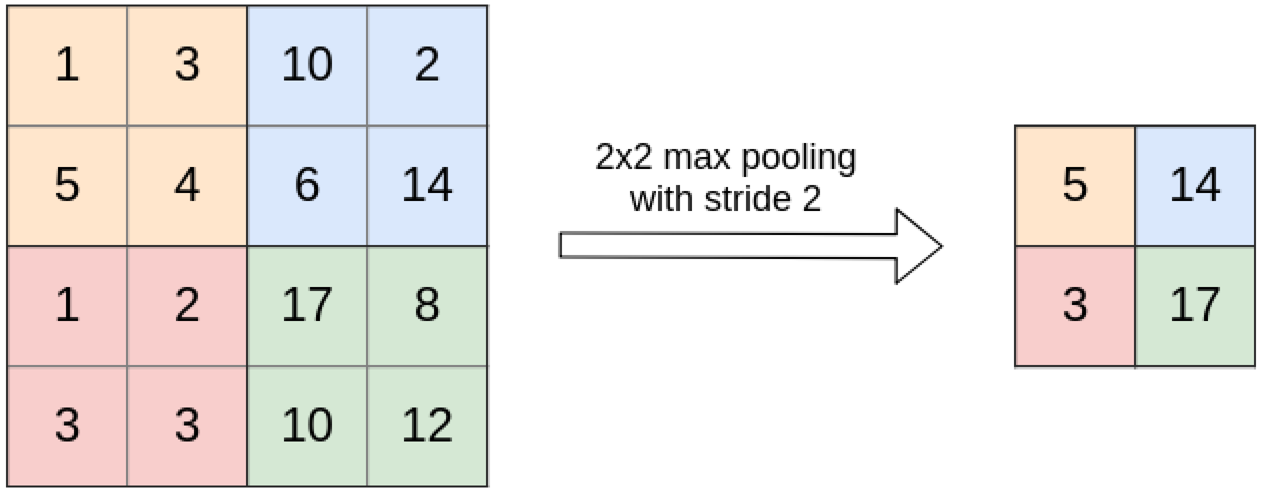
## Pooling layers

- In the previous section, we explained how to increase the receptive field of the neurons by using a stride larger than 1. But we can also do this with the help of **pooling layers**.
- A pooling layer splits the input slice into a grid, where each grid cell represents a receptive field of a number of neurons (just as a convolutional layer does). Then, a pooling operation is applied over each cell of the grid.
- Different types of pooling layers exist. Pooling layers don't change the volume depth, because the pooling operation is performed independently on each slice.

- There are two main types of pooling:
  - **Max pooling**
  - **Average pooling**
- Pooling layers don't have any weights.
- Pooling layers are defined by two parameters:
  - **Stride**, which is the same as with convolutional layers
  - **Receptive field size**, which is the equivalent of the filter size in convolutional layers

## Pooling layers

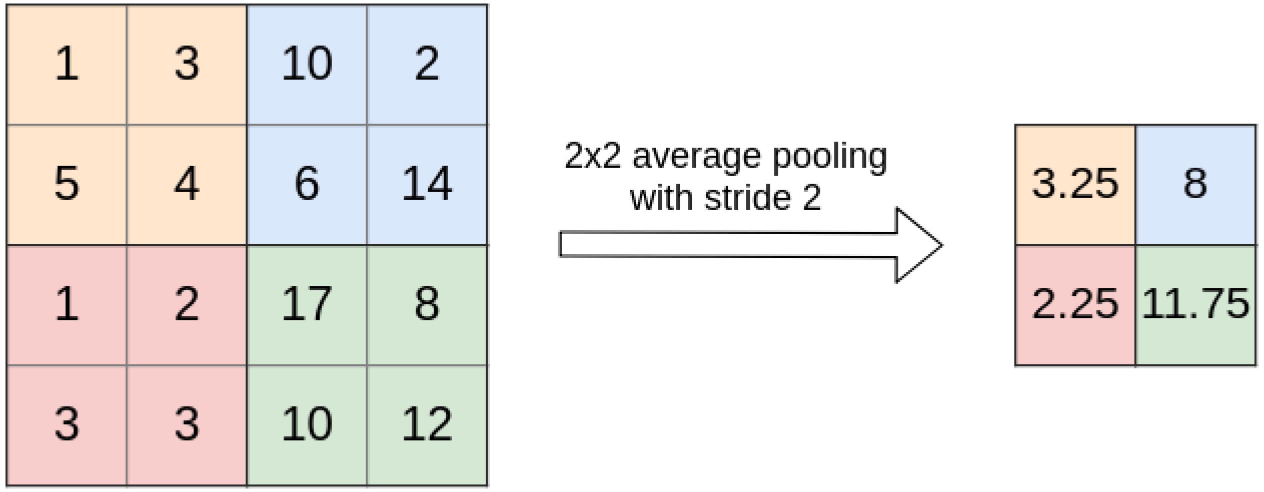
- **Max pooling:** is the most popular way of pooling. The max pooling operation takes the neuron with the highest activation value in each local receptive field (grid cell) and propagates only that value forward.



An example of the input and output of a max pooling operation with stride 2 and 2x2 receptive field. This operation discards 3/4 of the input neurons.

## Pooling layers

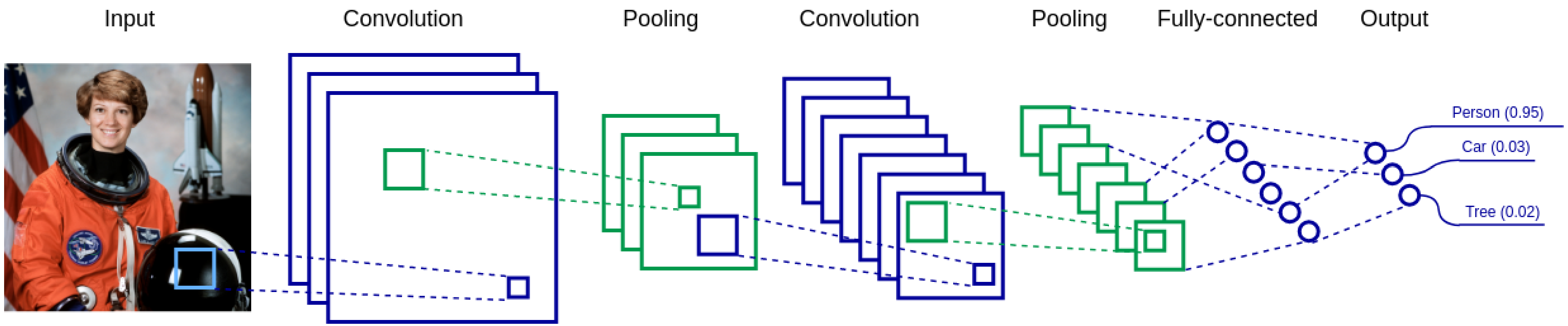
- **Average pooling:** is another type of pooling, where the output of each receptive field is the mean value of all activations within the field.



An example of the input and output of a max pooling operation with stride 2 and 2x2 receptive field

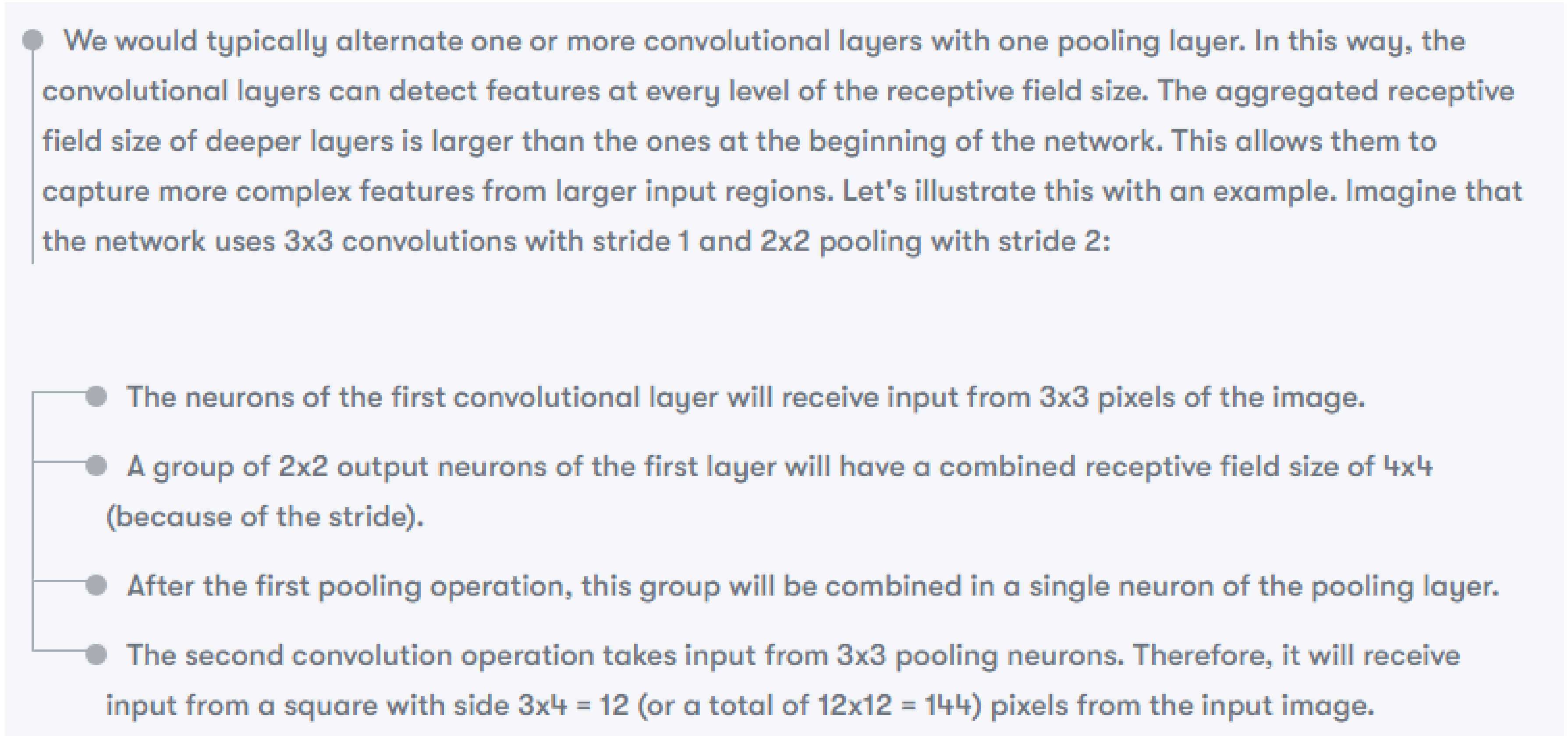
## The structure of a convolutional network

- A basic convolutional network with convolutional and fully-connected layers in blue and pooling layers in green.



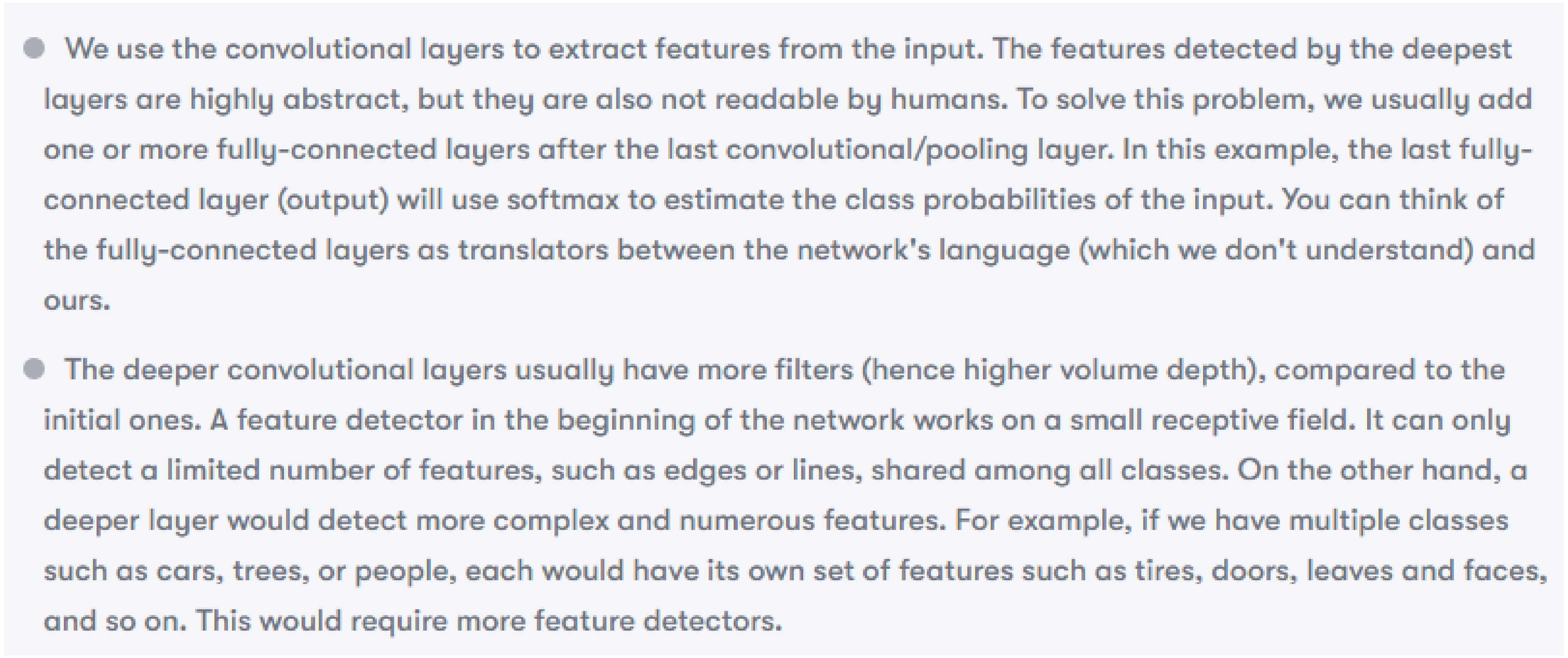
## The structure of a convolutional network

- Most CNNs share basic **properties**. Here are some of them:



- We would typically alternate one or more convolutional layers with one pooling layer. In this way, the convolutional layers can detect features at every level of the receptive field size. The aggregated receptive field size of deeper layers is larger than the ones at the beginning of the network. This allows them to capture more complex features from larger input regions. Let's illustrate this with an example. Imagine that the network uses 3x3 convolutions with stride 1 and 2x2 pooling with stride 2:
  - The neurons of the first convolutional layer will receive input from 3x3 pixels of the image.
  - A group of 2x2 output neurons of the first layer will have a combined receptive field size of 4x4 (because of the stride).
  - After the first pooling operation, this group will be combined in a single neuron of the pooling layer.
  - The second convolution operation takes input from 3x3 pooling neurons. Therefore, it will receive input from a square with side 3x4 = 12 (or a total of 12x12 = 144) pixels from the input image.

## The structure of a convolutional network

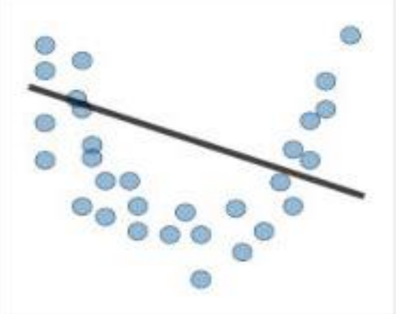
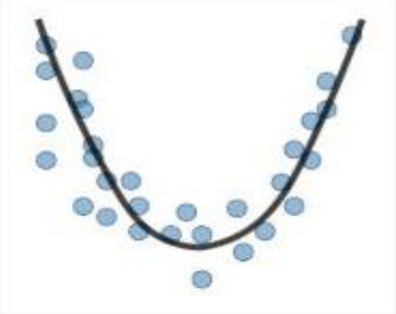
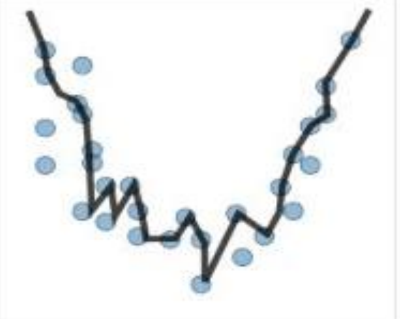
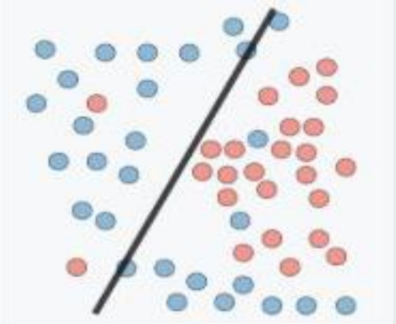
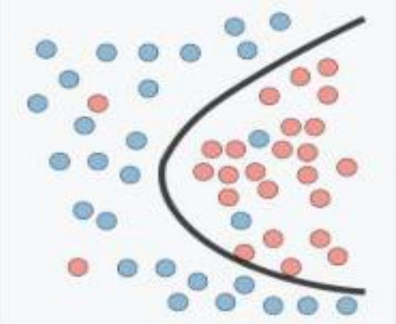
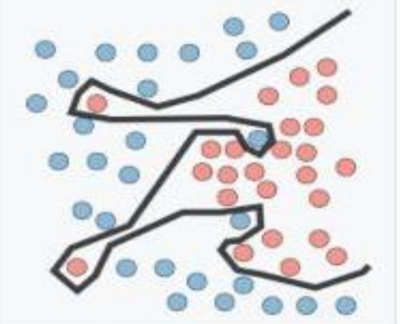
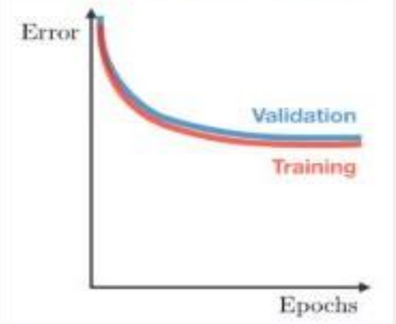
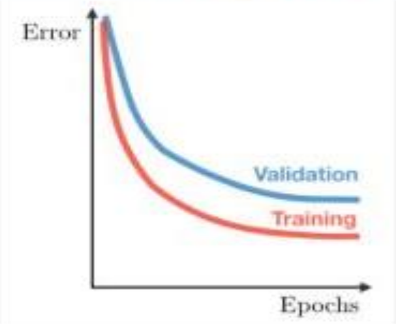
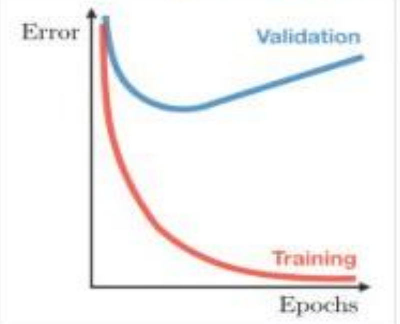


- We use the convolutional layers to extract features from the input. The features detected by the deepest layers are highly abstract, but they are also not readable by humans. To solve this problem, we usually add one or more fully-connected layers after the last convolutional/pooling layer. In this example, the last fully-connected layer (output) will use softmax to estimate the class probabilities of the input. You can think of the fully-connected layers as translators between the network's language (which we don't understand) and ours.
- The deeper convolutional layers usually have more filters (hence higher volume depth), compared to the initial ones. A feature detector in the beginning of the network works on a small receptive field. It can only detect a limited number of features, such as edges or lines, shared among all classes. On the other hand, a deeper layer would detect more complex and numerous features. For example, if we have multiple classes such as cars, trees, or people, each would have its own set of features such as tires, doors, leaves and faces, and so on. This would require more feature detectors.

Module (3)

## Improving the performance of CNNs

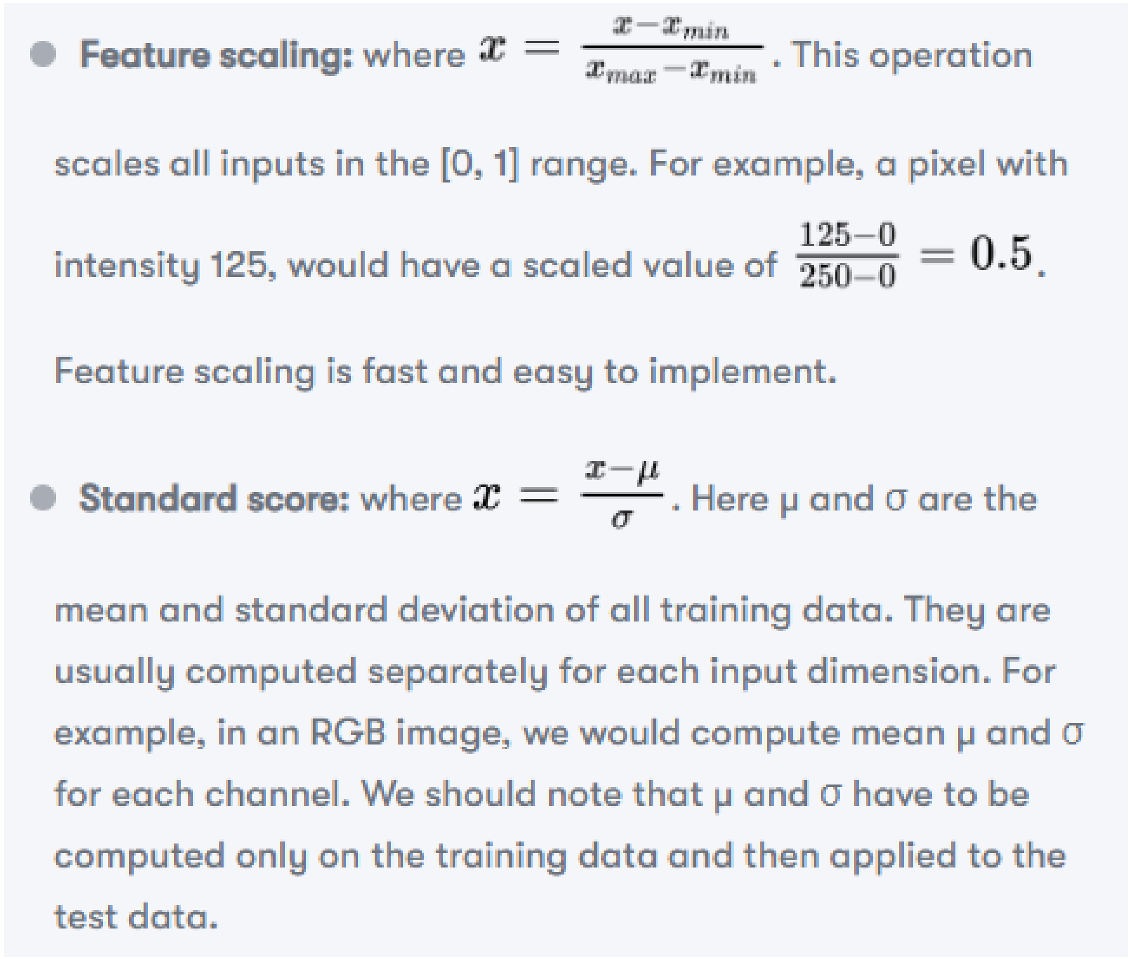
1. Data pre-processing
2. Regularization:
   - Weight decay
   - Dropout
   - Data augmentation
3. Batch normalization

| | Underfitting | Just right | Overfitting |
|---|---|---|---|
| **Symptoms** | • High training error<br>• Training error close to test error<br>• High bias | • Training error slightly lower than test error | • Very low training error<br>• Training error much lower than test error<br>• High variance |
| **Regression illustration** |  |  |  |
| **Classification illustration** |  |  |  |
| **Deep learning illustration** |  |  |  |
| **Possible remedies** | • Complexify model<br>• Add more features<br>• Train longer | | • Perform regularization<br>• Get more data |

## Improving the performance of CNNs

**1. Data pre-processing**

- Until now, we've fed the network with unmodified inputs. In the case of images, these are pixel intensities in the range [0:255]. But that's not optimal. Imagine an RGB image, where the intensities in one of the color channels is very high compared to the other two. When we feed the image to the network, the values of this channel will become dominant, diminishing the others. This could skew the results, because in reality every channel has equal importance.
- To solve this, we need to prepare (or **normalize**) the data, before we feed it to the network.
- In practice, we'll use two types of normalization →

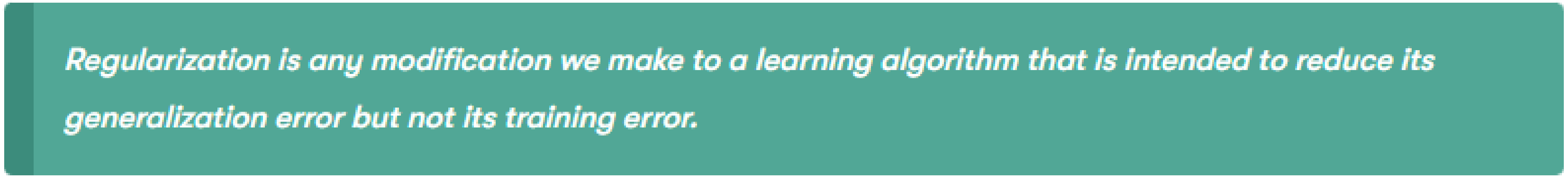


- **Feature scaling:** where $x = \frac{x - x_{min}}{x_{max} - x_{min}}$. This operation scales all inputs in the [0, 1] range. For example, a pixel with intensity 125, would have a scaled value of $\frac{125-0}{250-0} = 0.5$. Feature scaling is fast and easy to implement.
- **Standard score:** where $x = \frac{x - \mu}{\sigma}$. Here μ and σ are the mean and standard deviation of all training data. They are usually computed separately for each input dimension. For example, in an RGB image, we would compute mean μ and σ for each channel. We should note that μ and σ have to be computed only on the training data and then applied to the test data.

## Improving the performance of CNNs

**2. Regularization**

- We already know that overfitting is a central problem in machine learning (and even more so in deep networks).
- In this section, we'll discuss several ways to prevent it. Such techniques are collectively known as **regularization**.
- To quote Ian Goodfellow's Deep Learning book:



*Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

## Improving the performance of CNNs

### 2. Regularization → Weight decay

- The first technique we are going to discuss is weight decay (also known as L2 regularization).
- It works by adding additional terms to the value of the loss function. Without going into too much detail, we'll say that this term is a function of all the weights of the network. This means that, if the network weights have large values, the loss function increases. In effect, weight decay penalizes large network weights (hence the name). This prevents the network from relying too heavily on a few features associated with these weights. There is less chance of overfitting, when the network is forced to work with multiple features.
- In practical terms, we can add weight decay by changing the weight update rule, we introduced in Chapter 2, *Neural networks*, as shown in the following equations:

$$w \to w - \eta \nabla (J(w))$$

becomes

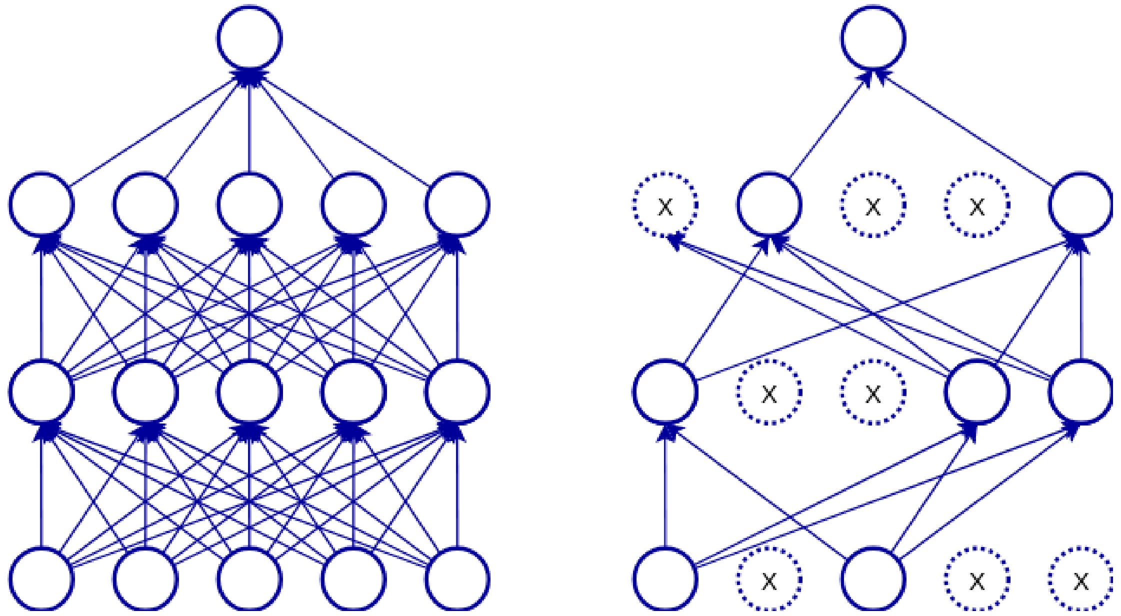
$$w \to w - \eta (\nabla (J(w)) - \lambda w)$$

where $\lambda$ is the weight decay coefficient.

## Improving the performance of CNNs

### 2. Regularization → Dropout

- Dropout is a regularization technique, which can be applied to the output of some of the network layers. Dropout randomly and periodically removes some of the neurons (along with their input and output connections) from the network. During a training mini-batch, each neuron has a probability *p* to be stochastically dropped. This is to ensure that no neuron ends up relying too much on other neurons and "learns" something useful for the network instead.
- Dropout can be applied after convolutional, pooling, or fully-connected layers.

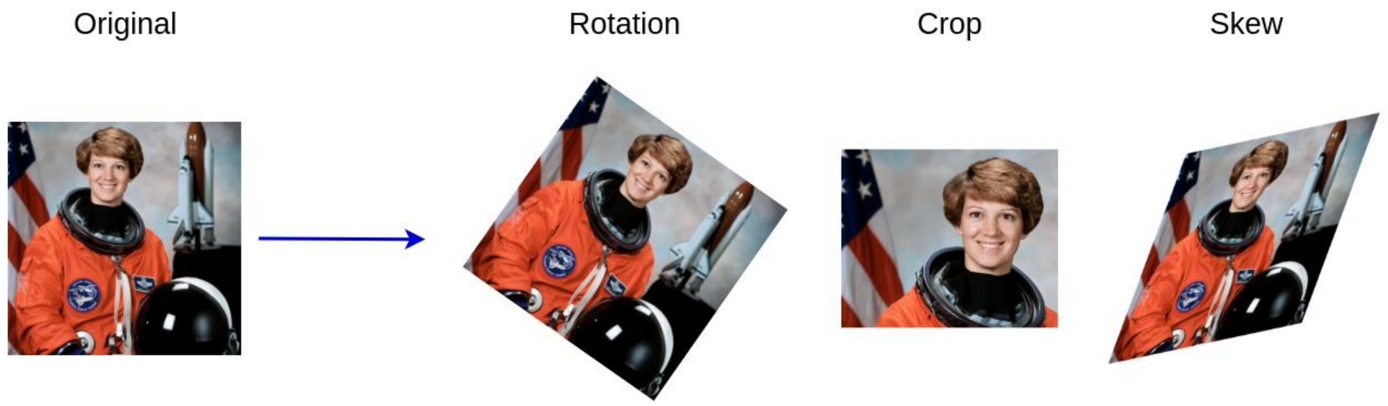


An example of dropout on fully-connected layers

## Improving the performance of CNNs

### 2. Regularization → Data augmentation

- One of the most efficient regularization techniques is data augmentation. If the training data is too small, the network might start to overfit. Data augmentation helps counter this by artificially increasing the size of the training set.
- Let's use an example. In the MNIST and CIFAR-10 examples, we've trained the network over multiple epochs. The network will "see" every sample of the dataset once per epoch. To prevent this, we can apply random augmentations to the images, before using them for training. The labels will stay the same. Some of the most popular image augmentations are:
  - **Rotation**
  - Horizontal and vertical flip
  - Zoom in/out
  - **Crop**
  - **Skew**
  - Contrast and brightness adjustment



## Improving the performance of CNNs

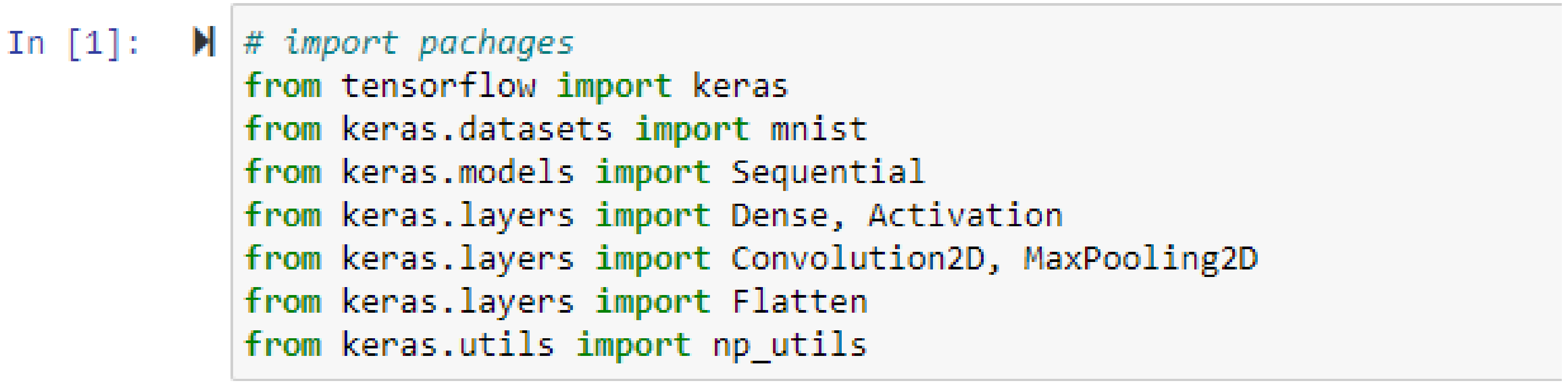
**3. Batch normalization**

- In *Data pre-processing*, we explained why data normalization is important. Batch normalization provides a way to apply data processing, similar to the standard score, for the hidden layers of the network.
- It normalizes the outputs of the hidden layer for each mini-batch (hence the name) in a way, which maintains its mean activation value close to 0, and its standard deviation close to 1.
- We can use it with both convolutional and fully-connected layers.
- Networks with batch normalization train faster and can use higher learning rates.
- For more information about batch normalization, see the original paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by Sergey Ioffe and Christian Szegedy, which can be seen at the following link: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

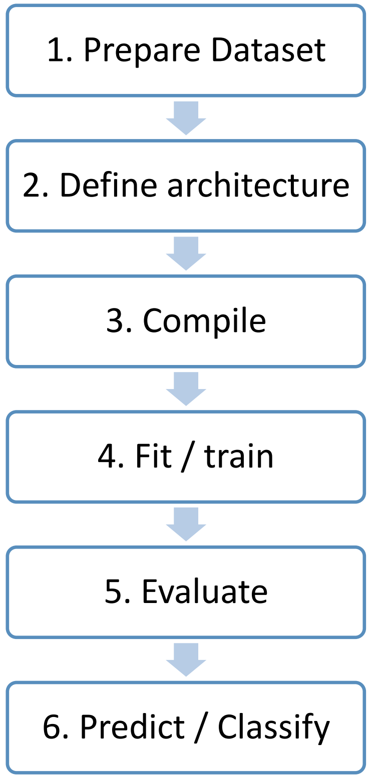
## Let's have fun ☺

# A CNN example with Keras and MNIST

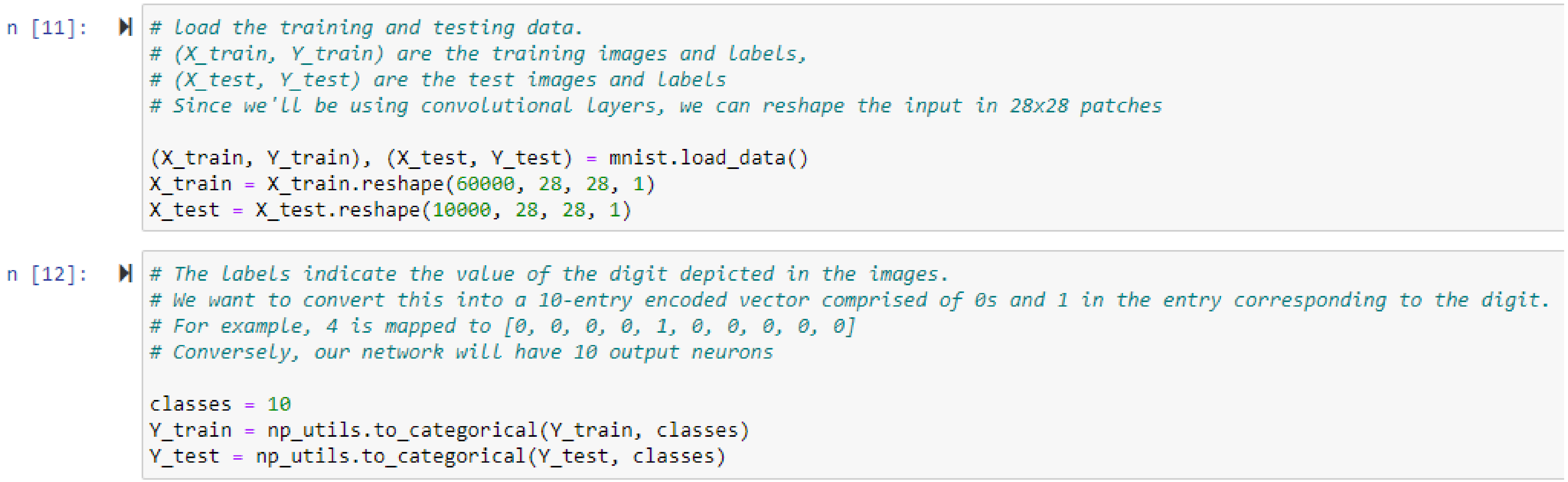
## 1. Prepare & Explore Dataset



```
In [1]:  # import pachages
         from tensorflow import keras
         from keras.datasets import mnist
         from keras.models import Sequential
         from keras.layers import Dense, Activation
         from keras.layers import Convolution2D, MaxPooling2D
         from keras.layers import Flatten
         from keras.utils import np_utils
```



## A CNN example with Keras and MNIST



n [11]:

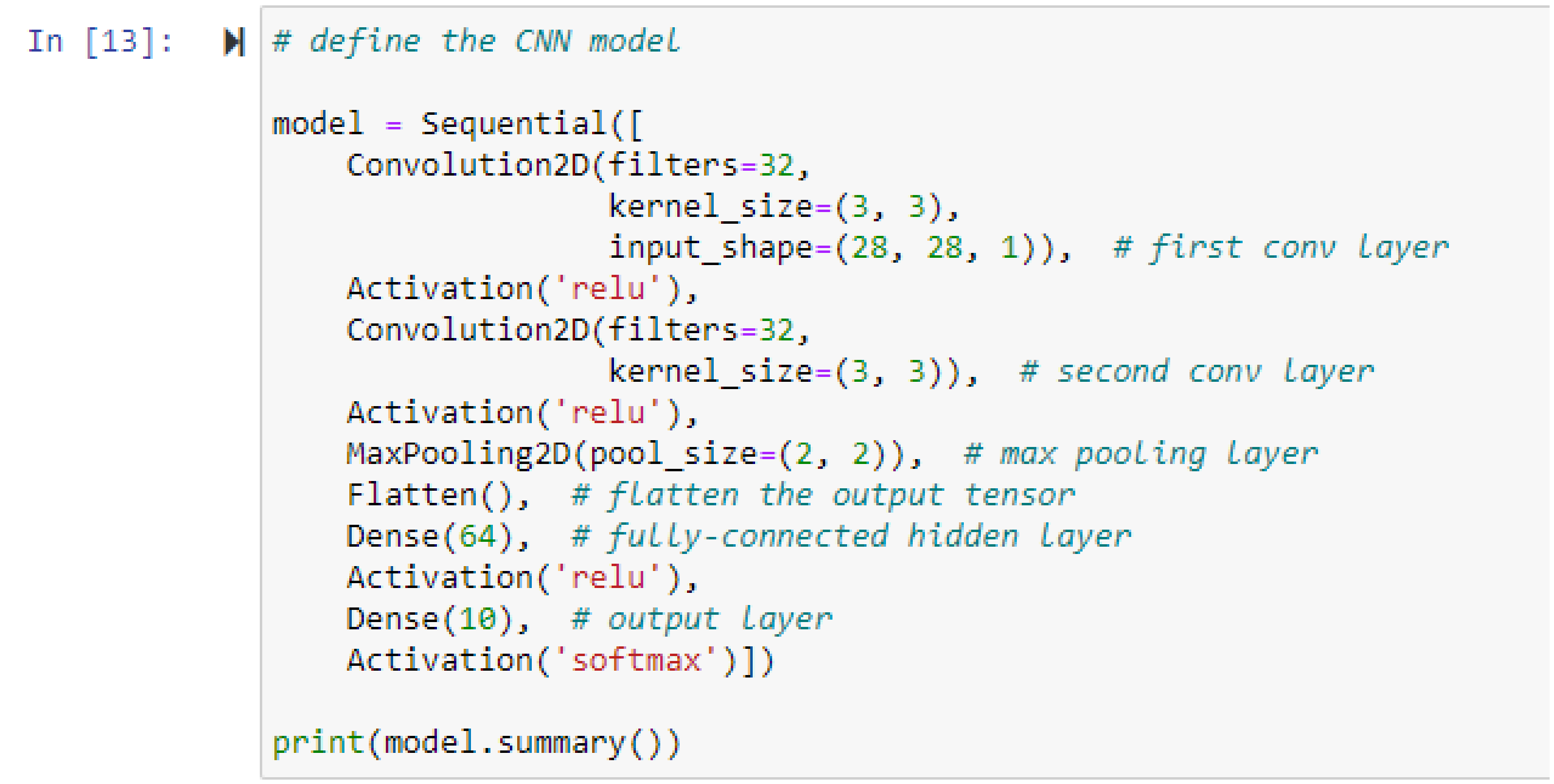
```
# Load the training and testing data.
# (X_train, Y_train) are the training images and labels,
# (X_test, Y_test) are the test images and labels
# Since we'll be using convolutional layers, we can reshape the input in 28x28 patches

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)
```

n [12]:

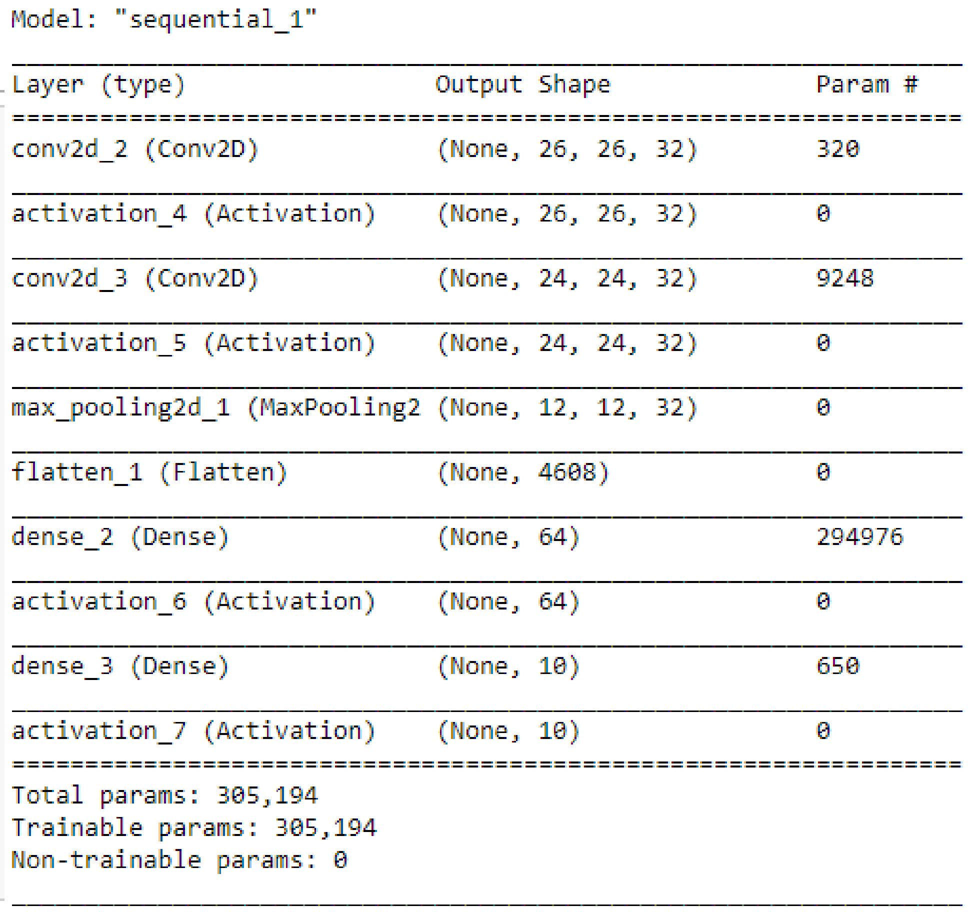
```
# The labels indicate the value of the digit depicted in the images.
# We want to convert this into a 10-entry encoded vector comprised of 0s and 1 in the entry corresponding to the digit.
# For example, 4 is mapped to [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
# Conversely, our network will have 10 output neurons

classes = 10
Y_train = np_utils.to_categorical(Y_train, classes)
Y_test = np_utils.to_categorical(Y_test, classes)
```

# A CNN example with Keras and MNIST

## 2. Define the neural network architecture

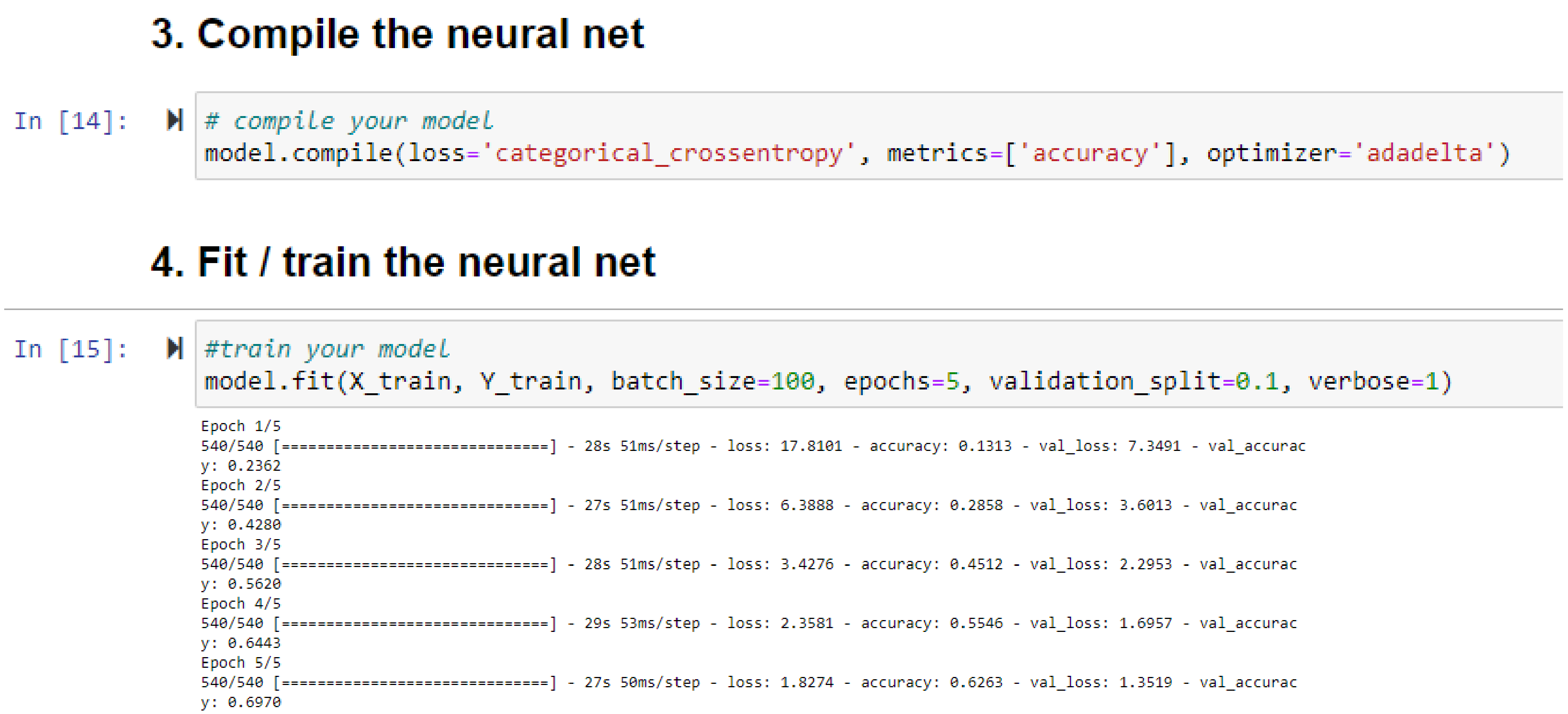


```
In [13]:  # define the CNN model

           model = Sequential([
               Convolution2D(filters=32,
                             kernel_size=(3, 3),
                             input_shape=(28, 28, 1)),   # first conv layer
               Activation('relu'),
               Convolution2D(filters=32,
                             kernel_size=(3, 3)),   # second conv layer
               Activation('relu'),
               MaxPooling2D(pool_size=(2, 2)),   # max pooling layer
               Flatten(),   # flatten the output tensor
               Dense(64),   # fully-connected hidden layer
               Activation('relu'),
               Dense(10),   # output layer
               Activation('softmax')])

           print(model.summary())
```



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
activation_4 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
activation_5 (Activation)    (None, 24, 24, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                294976
_________________________________________________________________
activation_6 (Activation)    (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
_________________________________________________________________
activation_7 (Activation)    (None, 10)                0
=================================================================
Total params: 305,194
Trainable params: 305,194
Non-trainable params: 0
_________________________________________________________________
```

## A CNN example with Keras and MNIST



### 3. Compile the neural net

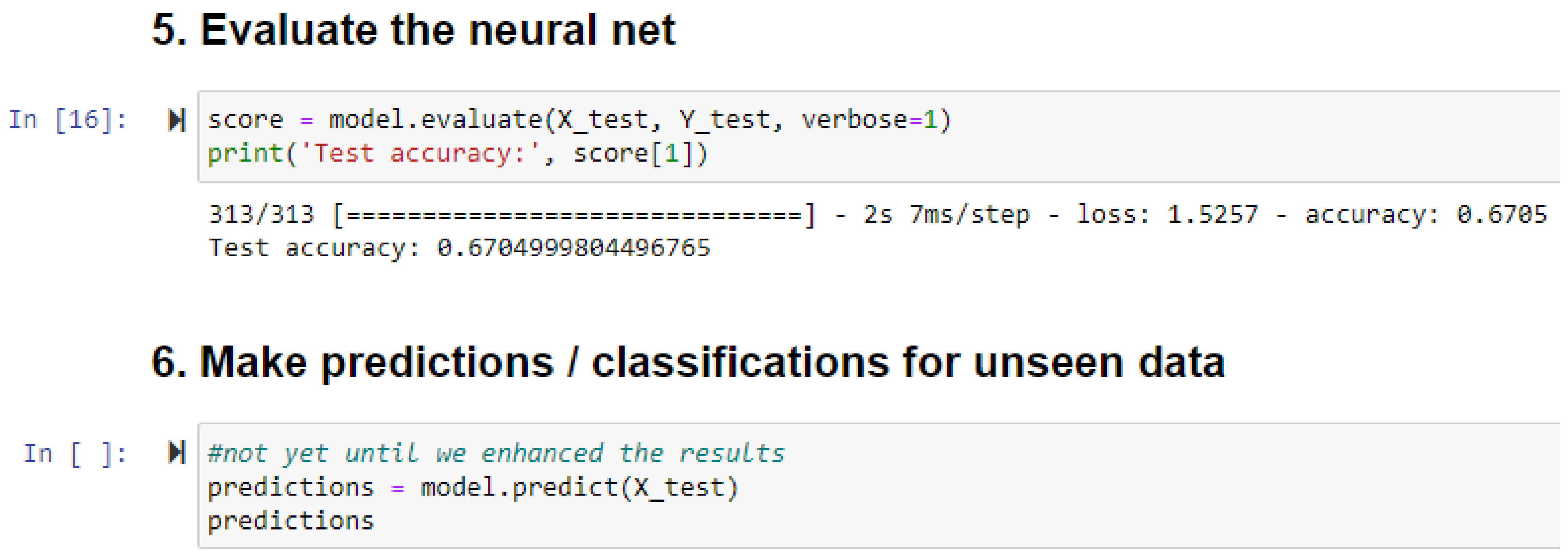
In [14]:
```
# compile your model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adadelta')
```

### 4. Fit / train the neural net

In [15]:
```
#train your model
model.fit(X_train, Y_train, batch_size=100, epochs=5, validation_split=0.1, verbose=1)
```

```
Epoch 1/5
540/540 [==============================] - 28s 51ms/step - loss: 17.8101 - accuracy: 0.1313 - val_loss: 7.3491 - val_accurac
y: 0.2362
Epoch 2/5
540/540 [==============================] - 27s 51ms/step - loss: 6.3888 - accuracy: 0.2858 - val_loss: 3.6013 - val_accurac
y: 0.4280
Epoch 3/5
540/540 [==============================] - 28s 51ms/step - loss: 3.4276 - accuracy: 0.4512 - val_loss: 2.2953 - val_accurac
y: 0.5620
Epoch 4/5
540/540 [==============================] - 29s 53ms/step - loss: 2.3581 - accuracy: 0.5546 - val_loss: 1.6957 - val_accurac
y: 0.6443
Epoch 5/5
540/540 [==============================] - 27s 50ms/step - loss: 1.8274 - accuracy: 0.6263 - val_loss: 1.3519 - val_accurac
y: 0.6970
```

## A CNN example with Keras and MNIST



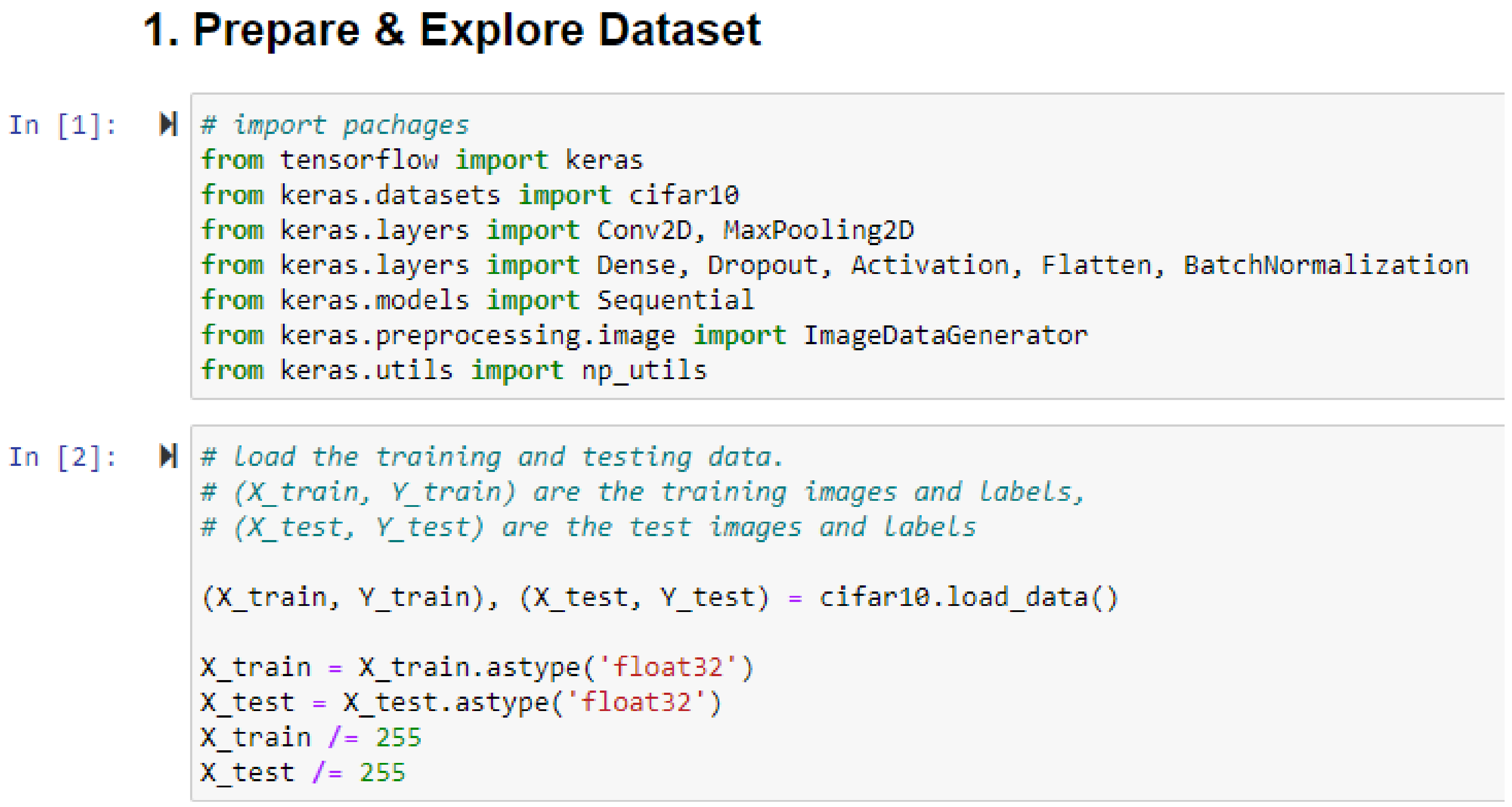
### 5. Evaluate the neural net

```
In [16]:  score = model.evaluate(X_test, Y_test, verbose=1)
          print('Test accuracy:', score[1])

          313/313 [==============================] - 2s 7ms/step - loss: 1.5257 - accuracy: 0.6705
          Test accuracy: 0.6704999804496765
```

### 6. Make predictions / classifications for unseen data

```
In [ ]:   #not yet until we enhanced the results
          predictions = model.predict(X_test)
          predictions
```

# A CNN example with Keras and CIFAR-10



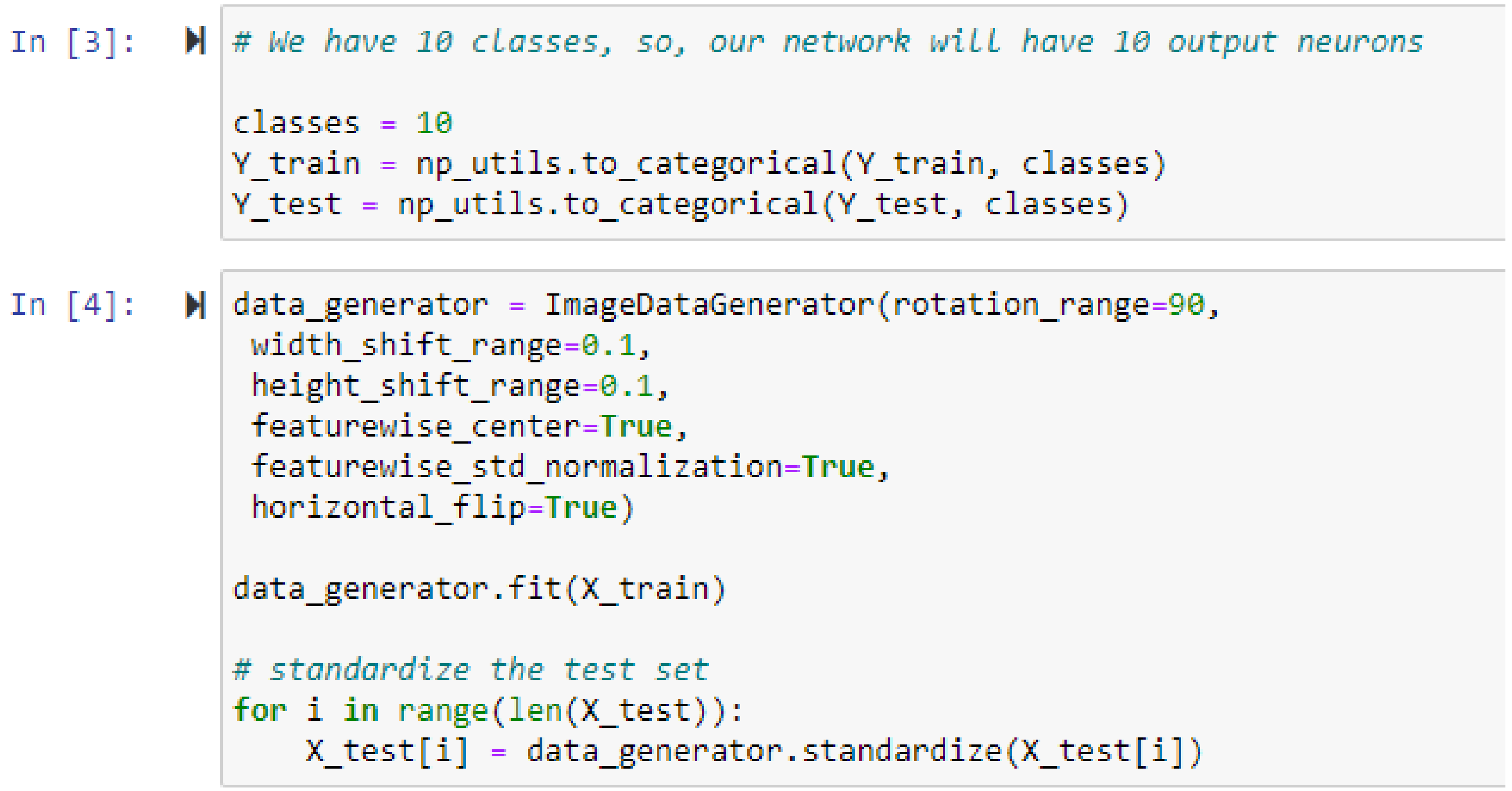
## 1. Prepare & Explore Dataset

In [1]:
```
# import pachages
from tensorflow import keras
from keras.datasets import cifar10
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import np_utils
```

In [2]:
```
# load the training and testing data.
# (X_train, Y_train) are the training images and labels,
# (X_test, Y_test) are the test images and labels

(X_train, Y_train), (X_test, Y_test) = cifar10.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

## A CNN example with Keras and CIFAR-10

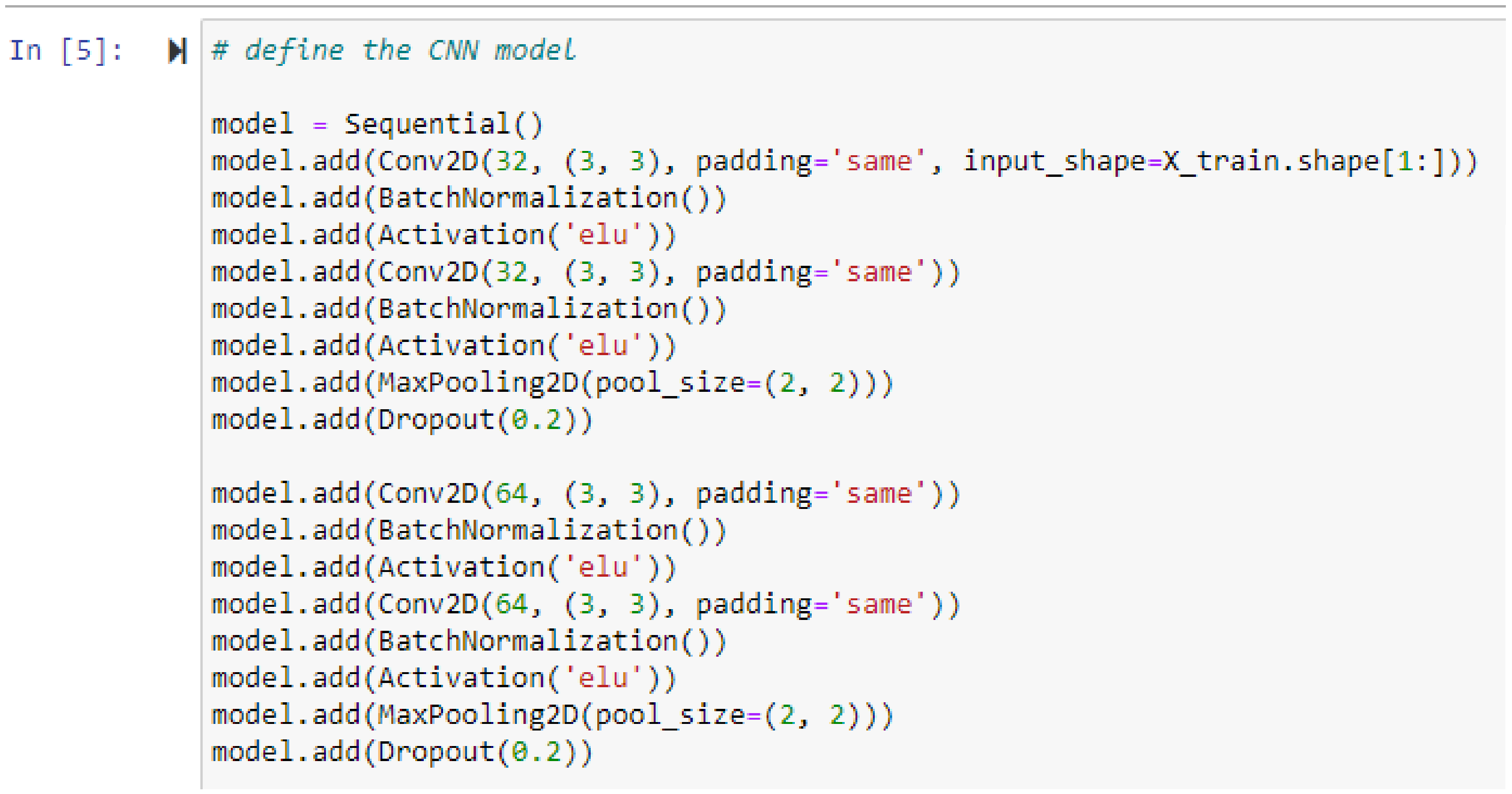


```
In [3]:  # We have 10 classes, so, our network will have 10 output neurons

         classes = 10
         Y_train = np_utils.to_categorical(Y_train, classes)
         Y_test = np_utils.to_categorical(Y_test, classes)
```

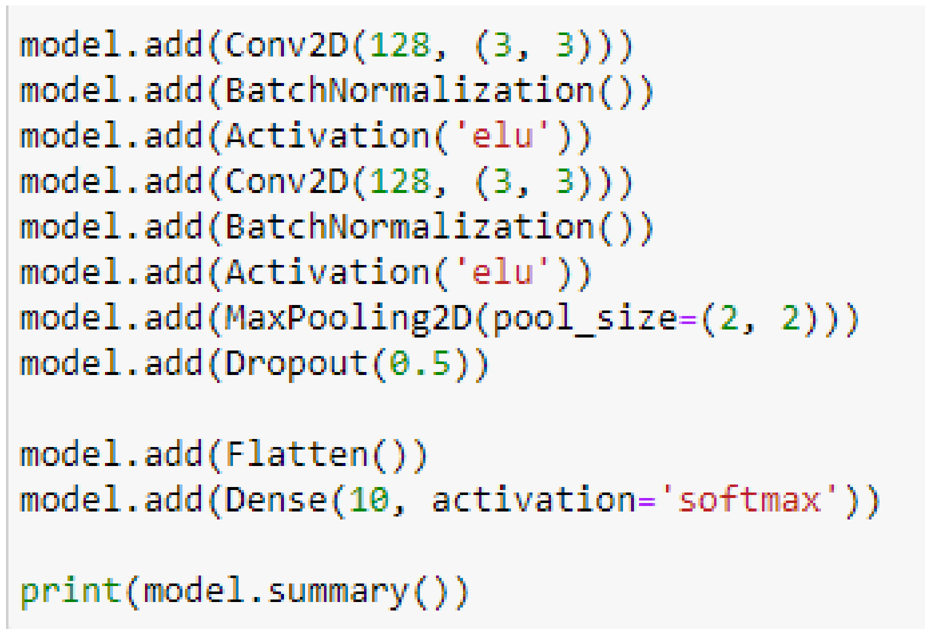
```
In [4]:  data_generator = ImageDataGenerator(rotation_range=90,
          width_shift_range=0.1,
          height_shift_range=0.1,
          featurewise_center=True,
          featurewise_std_normalization=True,
          horizontal_flip=True)

         data_generator.fit(X_train)

         # standardize the test set
         for i in range(len(X_test)):
             X_test[i] = data_generator.standardize(X_test[i])
```

# A CNN example with Keras and CIFAR-10

## 2. Define the neural network architecture



```
In [5]:  # define the CNN model

         model = Sequential()
         model.add(Conv2D(32, (3, 3), padding='same', input_shape=X_train.shape[1:]))
         model.add(BatchNormalization())
         model.add(Activation('elu'))
         model.add(Conv2D(32, (3, 3), padding='same'))
         model.add(BatchNormalization())
         model.add(Activation('elu'))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Dropout(0.2))

         model.add(Conv2D(64, (3, 3), padding='same'))
         model.add(BatchNormalization())
         model.add(Activation('elu'))
         model.add(Conv2D(64, (3, 3), padding='same'))
         model.add(BatchNormalization())
         model.add(Activation('elu'))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Dropout(0.2))
```



```
model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

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

print(model.summary())
```

## A CNN example with Keras and CIFAR-10

### 3. Compile the neural net

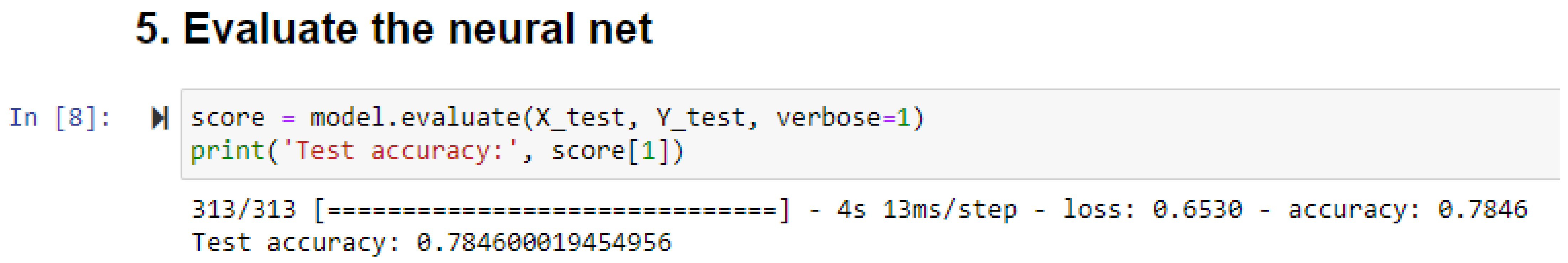
```
In [6]:  # compile your model
         model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

### 4. Fit / train the neural net

```
In [7]:  #train your model
         batch_size = 50
         model.fit_generator(
          generator=data_generator.flow(x=X_train,
          y=Y_train,
          batch_size=batch_size),
          steps_per_epoch=len(X_train) // batch_size,
          epochs=100,
          validation_data=(X_test, Y_test),
          workers=4)
```

```
Epoch 100/100
1000/1000 [==============================] - 148s 148ms/step - loss: 0.6312 - accuracy: 0.7812 - val_loss: 0.6530 - val_a
ccuracy: 0.7846
```

## A CNN example with Keras and CIFAR-10



### 5. Evaluate the neural net

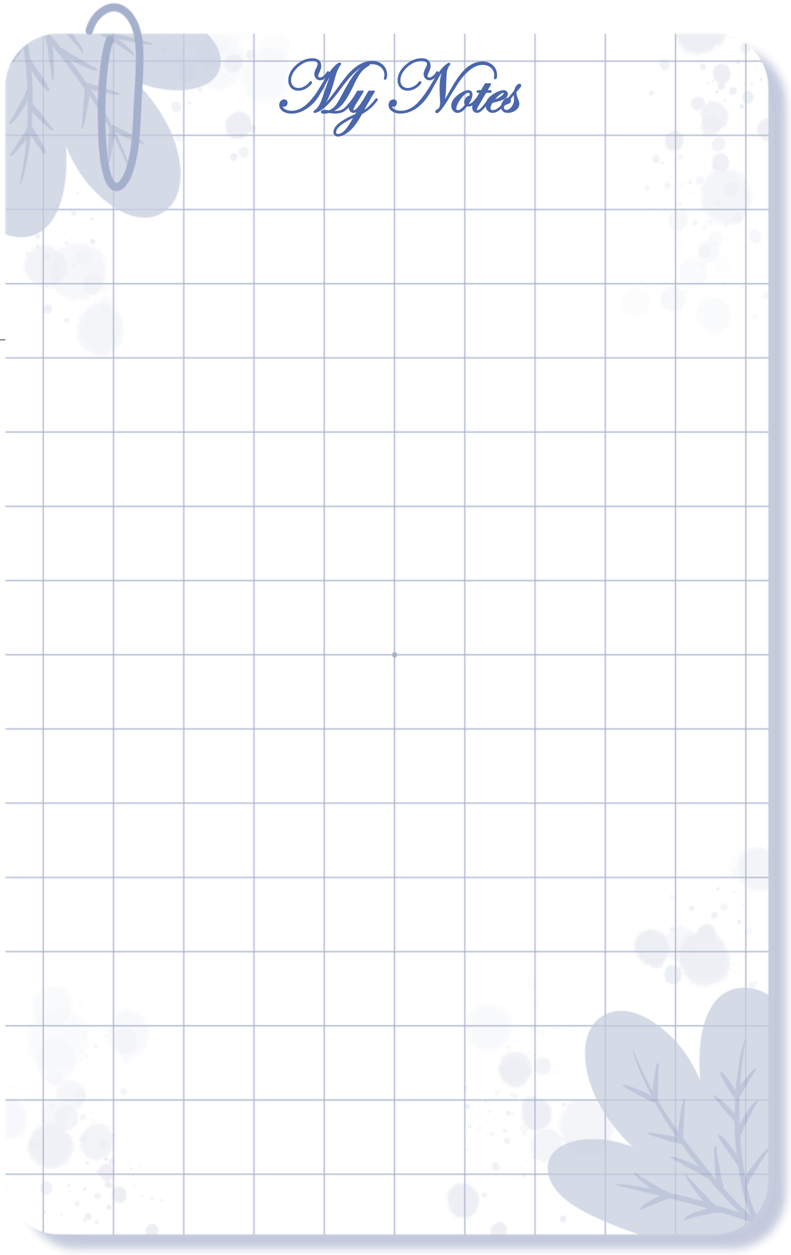
In [8]:

```
score = model.evaluate(X_test, Y_test, verbose=1)
print('Test accuracy:', score[1])
```

```
313/313 [==============================] - 4s 13ms/step - loss: 0.6530 - accuracy: 0.7846
Test accuracy: 0.784600019454956
```

## Recap!

- ☑ Intuition and justification for CNNs
- ☑ Convolutional layers
- ☑ Stride and padding
- ☑ Pooling layers
- ☑ The structure of a convolutional network
- ☑ Improving the performance of CNNs
- ☑ A CNN example with Keras and MNIST
- ☑ A CNN example with Keras and CIFAR-10



All successful people men and women are big dreamers. They imagine what their future could be, ideal in every respect, and then they work every day toward their distant vision, that goal or purpose.

Brian Tracy

@SalhaAlzahrani