1. ***Image classification using MLP:***

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).

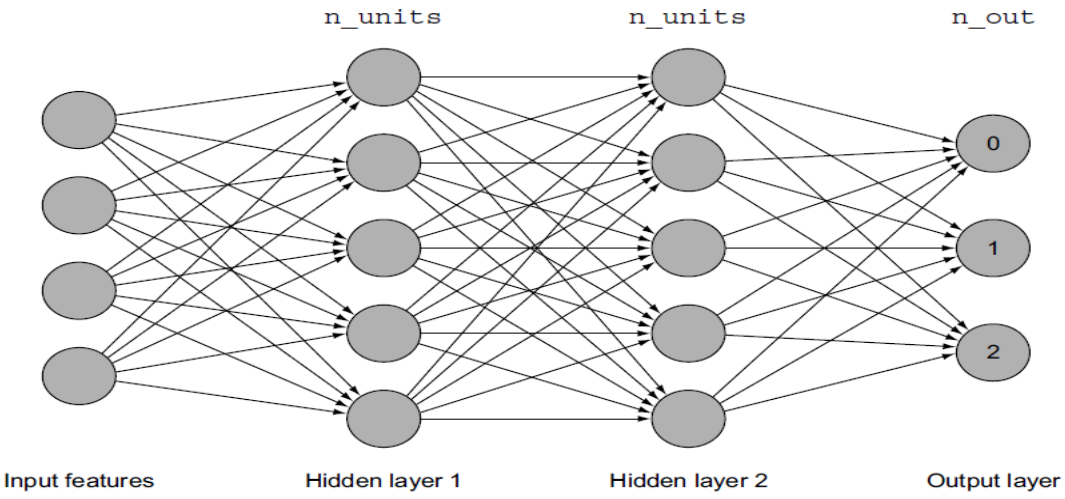


Figure 3.1 The MLP architecture consists of layers of neurons connected by weight connections.

- ***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).

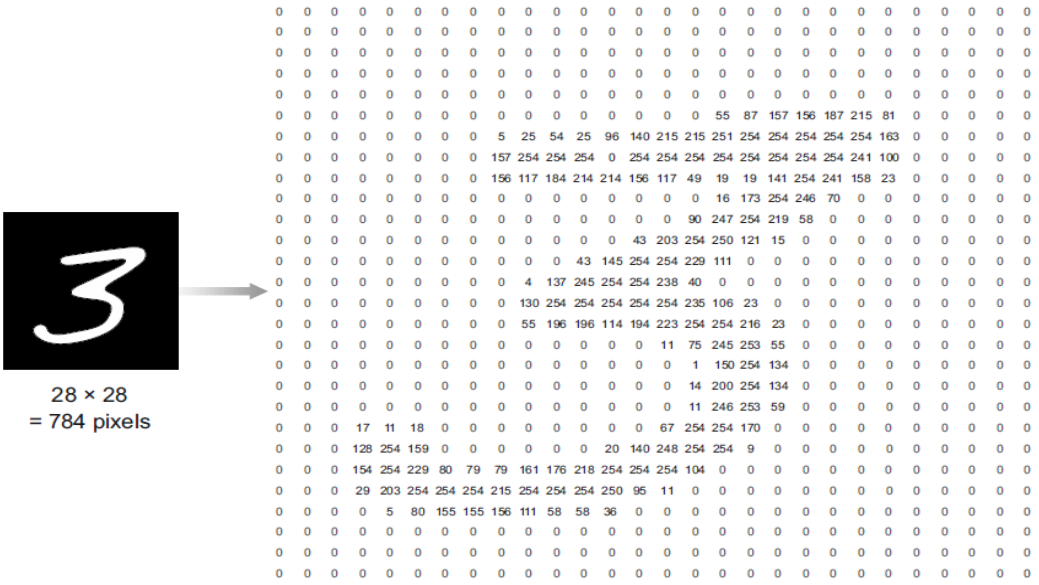


Figure 3.2 The computer sees this image as a 28 × 28 matrix of pixel values ranging from 0 to 255.

Since MLPs only take as input 1D vectors with dimensions (1, *n*), they cannot take a raw 2D image

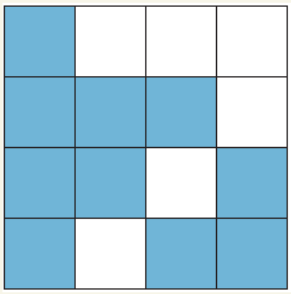
matrix with dimensions (*x*, *y*). To fit the image in the input layer, we first need to transform our image into one large vector with the dimensions (1, *n*) that contains all the pixel values in the image. This process is called *image flattening*. In this example, the total number (*n*) of pixels 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 = [*row*1, *row*2, *row*3, ..., *row*28]

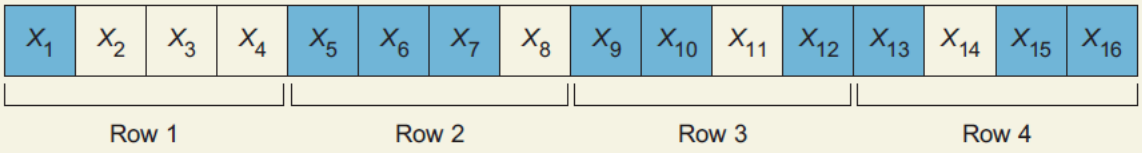
That said, the input layer in this example will have a total of 784 nodes: *x*1, *x*2, …, *x*784.

Visualizing input vectors

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



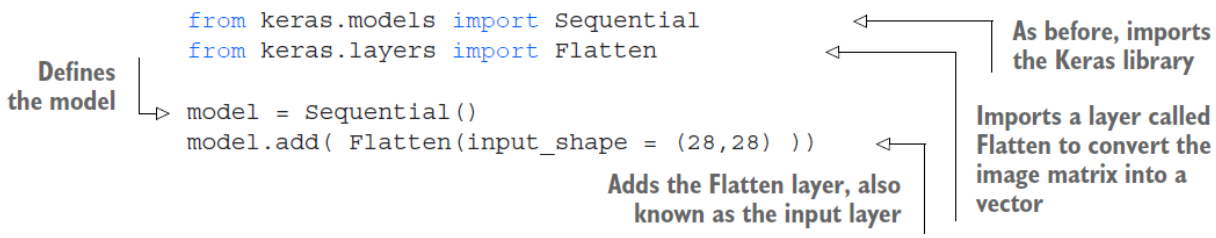
The input (*x*) is a flattened vector with the dimensions (1, 16):



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 the input image in Keras:



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.

- ***Hidden layers:***

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.

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
```

- **Output-layer:**

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'))
```

- **Putting it all together:**

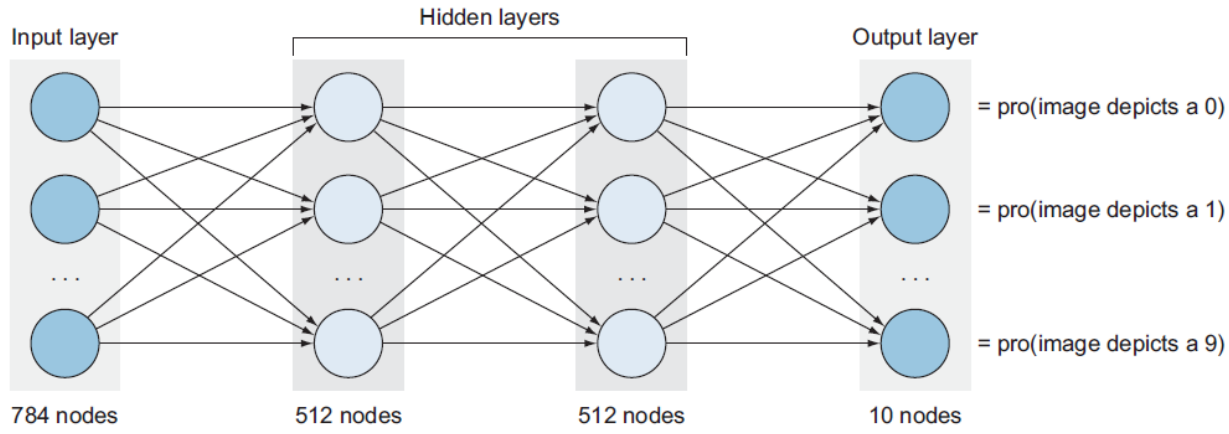
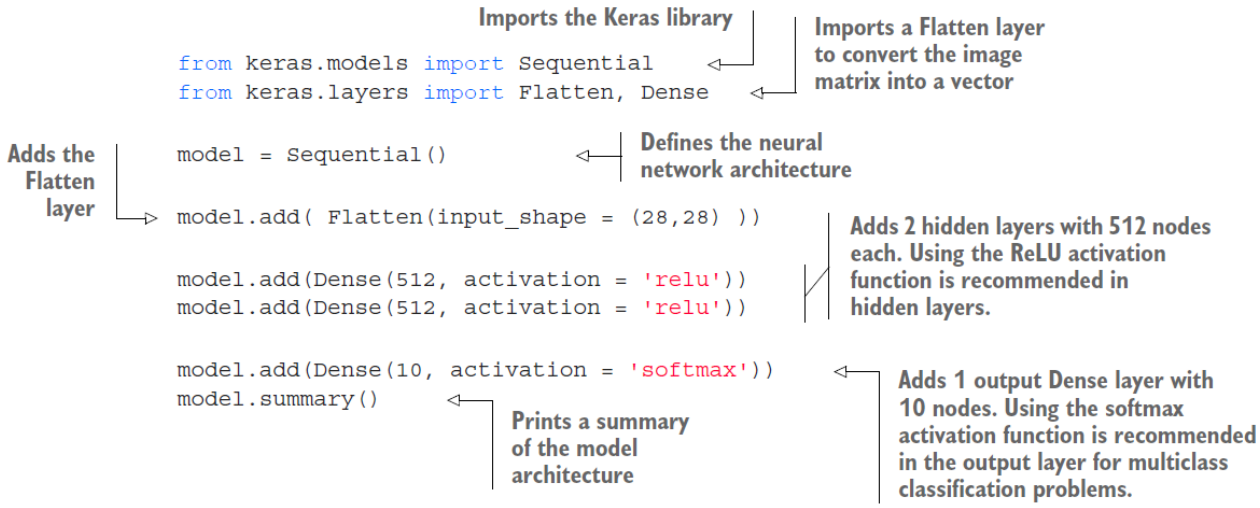


Figure 3.3 The neural network we create by combining the input, hidden, and output layers

Here is how it looks in Keras:



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.

Drawbacks of MLPs for processing images:

The two major problems in MLPs that convolutional networks are designed to fix are:
SPATIAL FEATURE LOSS and FULLY CONNECTED (DENSE) LAYERS

a. SPATIAL FEATURE LOSS

Flattening a 2D image to a 1D vector input results in losing the spatial features of the image. 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.

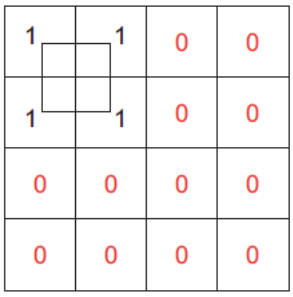


Figure 3.5 If the pixel value 1 is white and 0 is black, this is what our matrix looks like for identifying a square.

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 $x1$, $x2$, $x5$, and $x6$ 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?

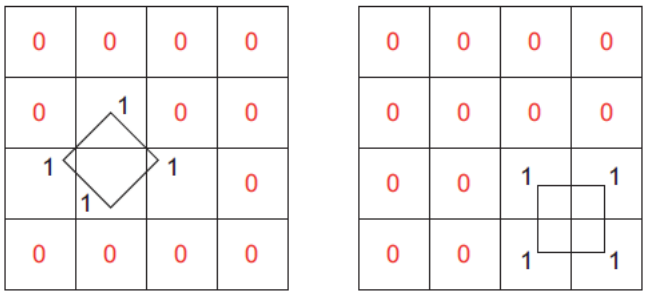


Figure 3.6 Square shapes in different areas of the image

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.

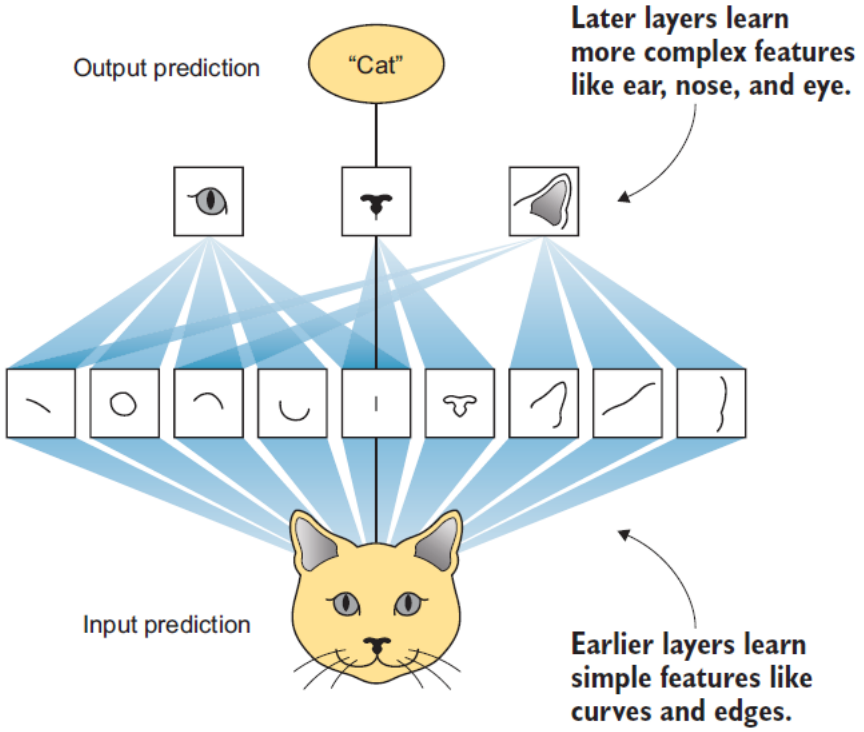


Figure 3.7 CNNs learn the image features through its layers.

b. 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.

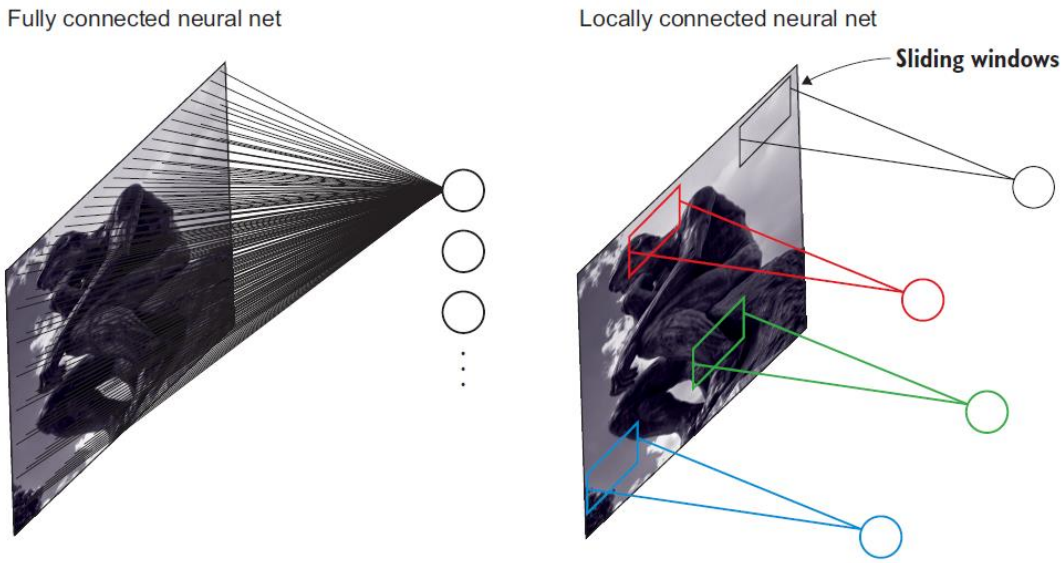


Figure 3.8 (Left) Fully connected neural network where all neurons are connected to all pixels of the image. (Right) Locally connected network where only a subset of pixels is connected to each neuron. These subsets are called *sliding windows*.

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.

1. CNN Architecture:

A Convolutional Neural Network (CNN) is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

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 below).

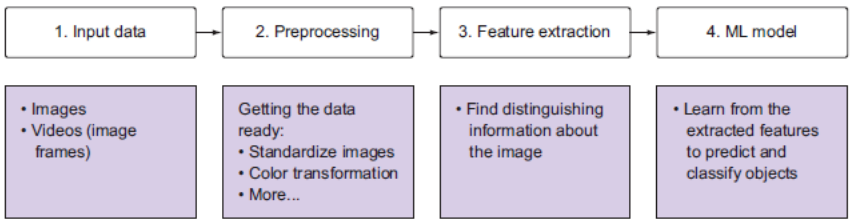


Figure 3.9 The image classification pipeline consists of four components: data input, data preprocessing, feature extraction, and the ML algorithm.

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.

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:

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

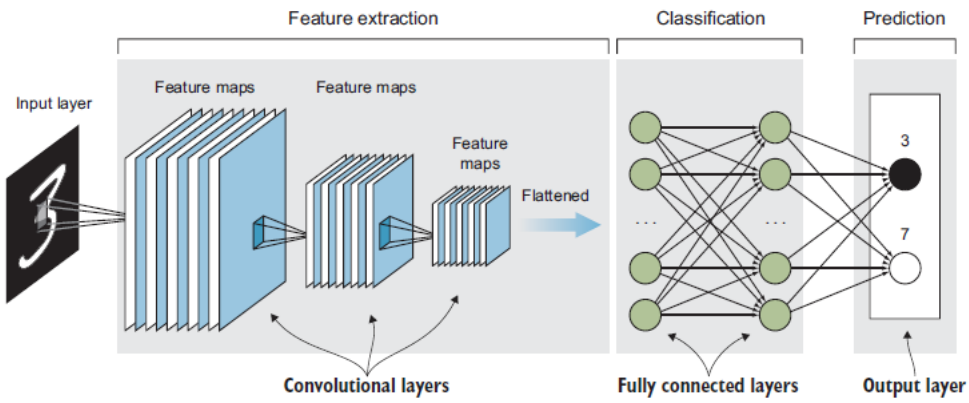


Figure 3.10 The CNN architecture consists of the following: input layer, convolutional layers, fully connected layers, and output prediction.

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.

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 below). 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.

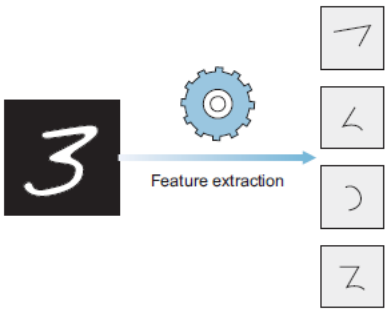


Figure 3.11 An image is broken into smaller images that contain distinctive features.

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.

let’s look at the second feature, 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.

Basic components of a CNN:
The three main components of a CNN architecture as shown in figure below are:
1 Convolutional layer (CONV)
2 Pooling layer (POOL)
3 Fully connected layer (FC)

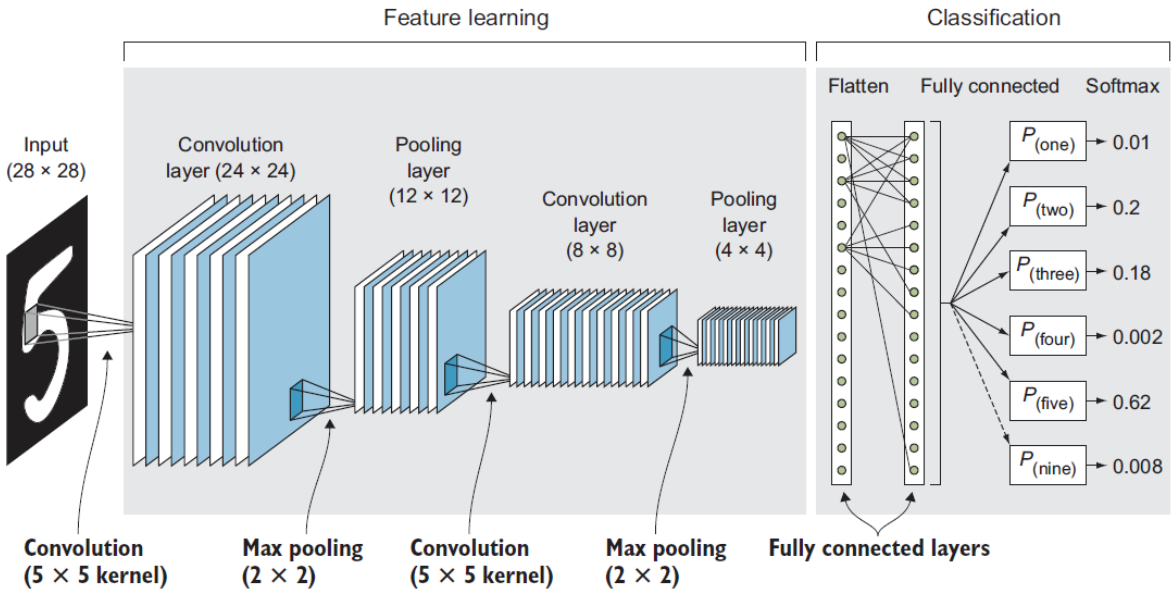


Figure 3.12 The basic components of convolutional networks are convolutional layers and pooling layers to perform feature extraction, and fully connected layers for classification.

1 Convolutional layer (CONV):
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.

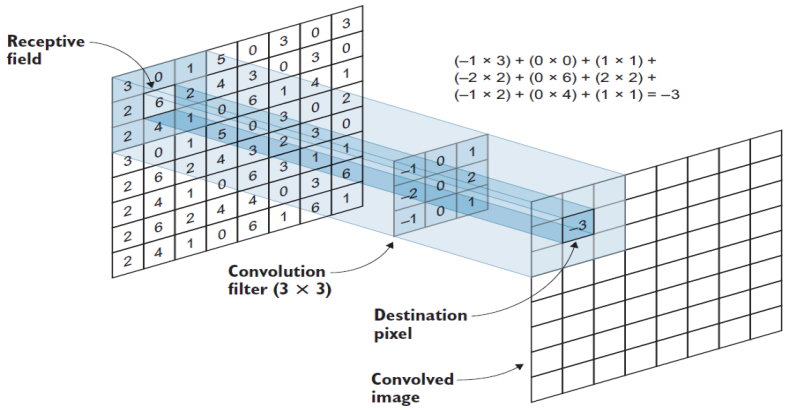


Figure 3.13 A 3 × 3 convolutional filter is sliding over the input image.

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).

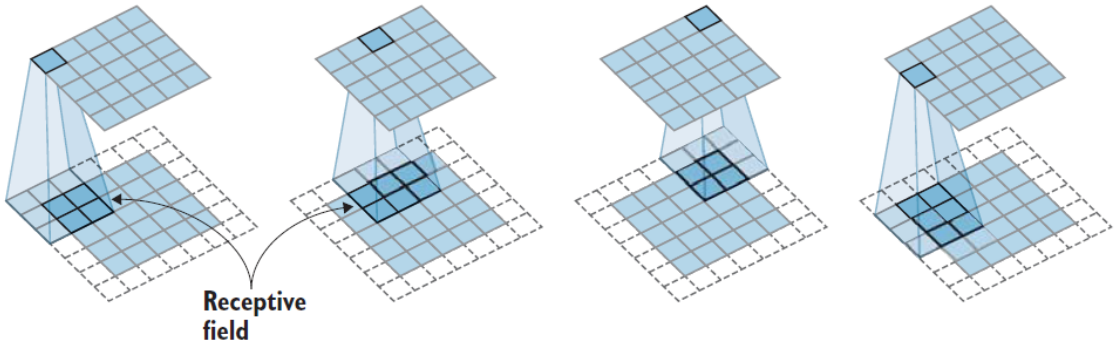


Figure 3.14 The kernel slides over the original image pixel by pixel and calculates the convolved image on the next layer. The convolved area is called the *receptive field*.

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

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

We 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).

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

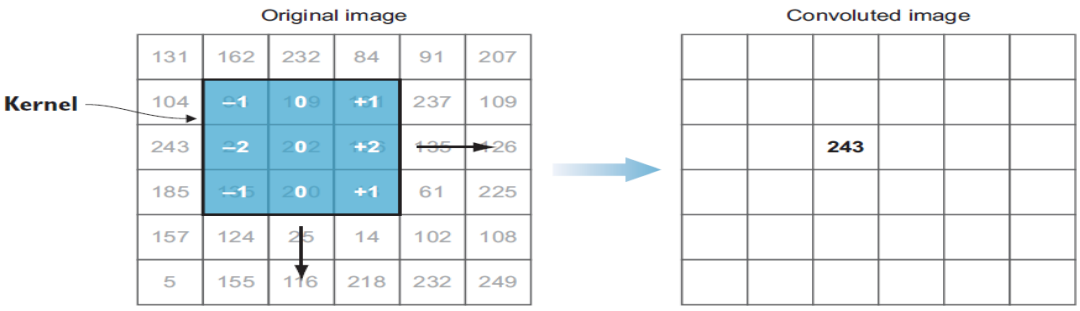


Figure 3.15 Multiplying each pixel in the receptive field by the corresponding pixel in the convolution filter and summing them gives the value of the center pixel in the new image.

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*.

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'))
```

Four hyperparameters (hyperparameters are the knobs you tune (increase and decrease) when configuring your neural network to improve performance) that control the size and depth of the output volume:

- **Filters:** Convolution filter is also known as a kernel. A filter, or kernel, in a CNN is a small matrix of weights that slides over the input data (such as an image), performs element-wise multiplication with the part of the input it is currently on, and then sums up all the results into a single output pixel. This process is known as convolution.
- **Kernel size:** is the size of the convolutional filter matrix. Sizes vary: 2 × 2, 3 × 3, 5 × 5.

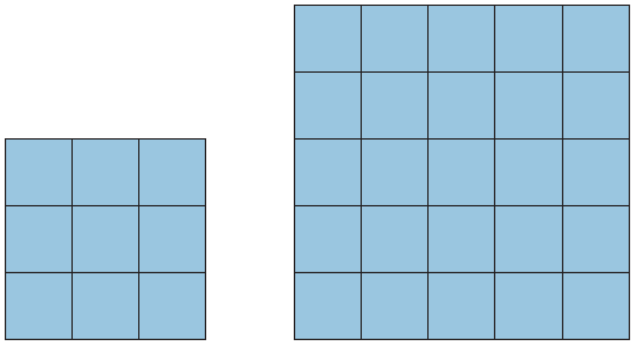


Figure 3.18 The kernel size refers to the dimensions of the convolution filter.

- **Stride:** 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.

Stride Size Definition:

The stride size is the number of pixels the convolutional kernel moves at each step or slide across the input data. A stride of 1 means the kernel moves one pixel at a time, while a larger stride (e.g., 2) means the kernel skips pixels, resulting in a larger step.

Effect on Feature Map Size:

Smaller stride sizes result in larger feature maps because the convolutional kernel overlaps more and covers the input data more comprehensively. Larger stride sizes result in smaller feature maps because the convolutional kernel skips over parts of the input, reducing the coverage.

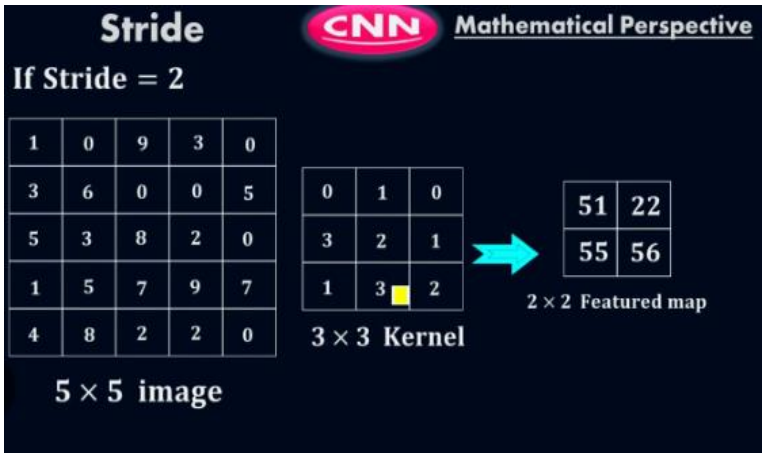
Mathematical Relationship:

The size of the feature map (output size) can be computed using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data (e.g., width or height), "Kernel size" is the size of the convolutional kernel, and "Stride" is the stride size.

Example:



- **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.

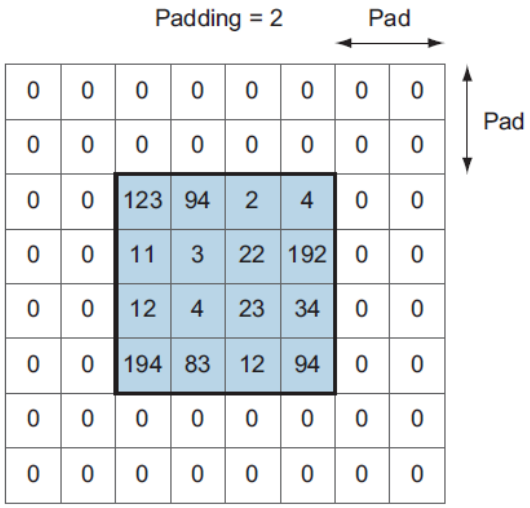


Figure 3.19 Zero-padding adds zeros around the border of the image. Padding = 2 adds two layers of zeros around the border.

Mathematical Representation:

The size of the feature map after convolution (or pooling) can be calculated using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data, "Kernel size" is the size of the convolutional kernel, "Padding" is the size of the zero-padding, and "Stride" is the stride size.

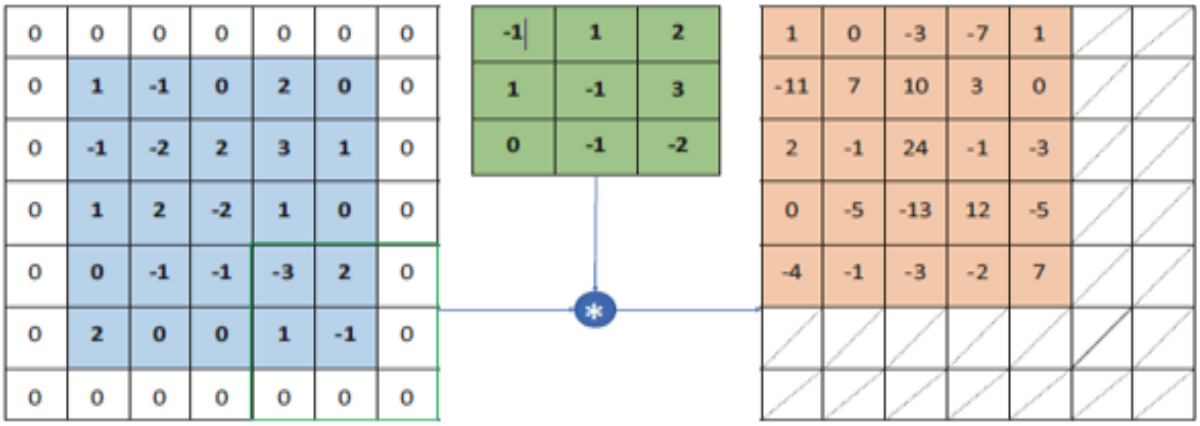
$$h' = \left\lfloor \frac{h + 2 \cdot p_h - k}{s} \right\rfloor + 1$$

$$w' = \left\lfloor \frac{w + 2 \cdot p_w - k}{s} \right\rfloor + 1$$

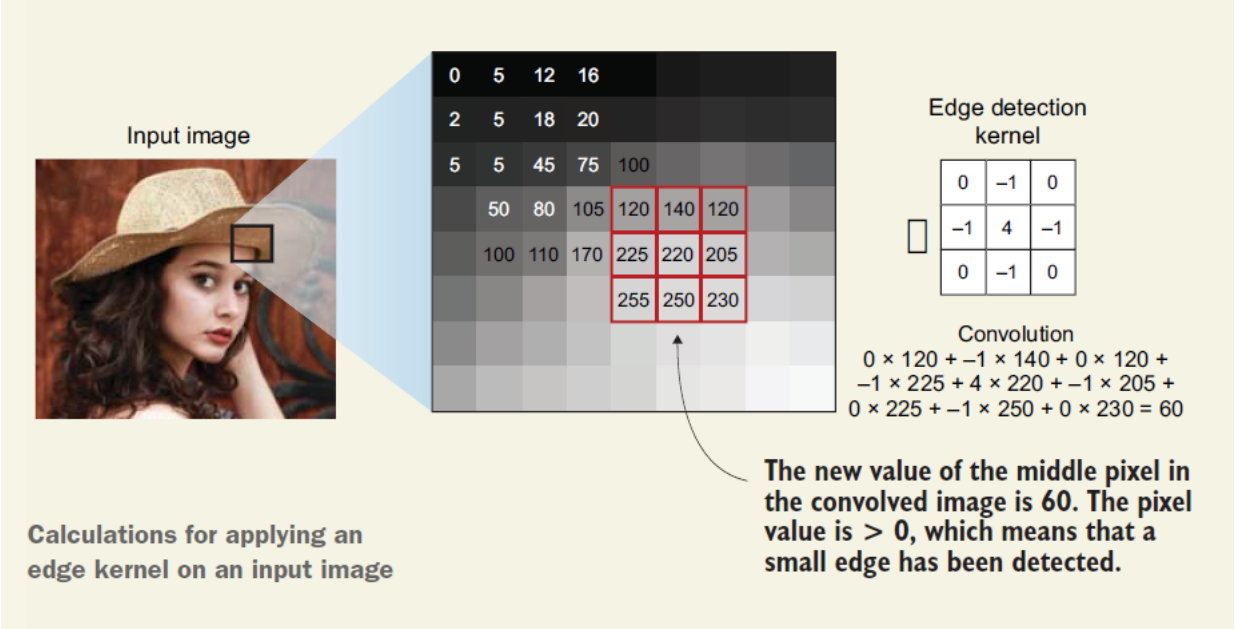
$$h' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$

$$w' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$

Taking the preceding example as an example, $h = w = 5$, $k = 3$, $ph = pw = 1$, $s = 1$, the output are:



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

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'))
```

Four hyperparameters (hyperparameters are the knobs you tune (increase and decrease) when configuring your neural network to improve performance) that control the size and depth of the output volume:

- **Filters:** Convolution filter is also known as a kernel. A filter, or kernel, in a CNN is a small matrix of weights that slides over the input data (such as an image), performs element-wise multiplication with the part of the input it is currently on, and then sums up all the results into a single output pixel. This process is known as convolution.
- **Kernel size:** It controls the size of the receptive field and how much of the input is considered at once. It is the size of the convolutional filter matrix. Sizes vary: 2 × 2, 3 × 3, 5 × 5.

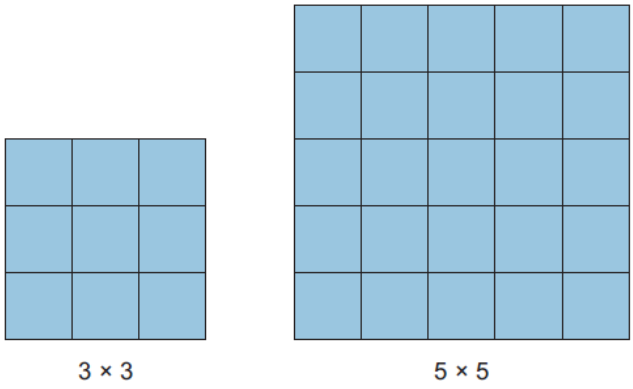


Figure 3.18 The kernel size refers to the dimensions of the convolution filter.

- **Stride:** 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.

Stride Size Definition:

The stride size is the number of pixels the convolutional kernel moves at each step or slide across the input data. A stride of 1 means the kernel moves one pixel at a time, while a larger stride (e.g., 2) means the kernel skips pixels, resulting in a larger step.

Effect on Feature Map Size:

Smaller stride sizes result in larger feature maps because the convolutional kernel overlaps more and covers the input data more comprehensively. Larger stride sizes result in smaller feature maps because the convolutional kernel skips over parts of the input, reducing the coverage.

Mathematical Relationship:

The size of the feature map (output size) can be computed using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data (e.g., width or height), "Kernel size" is the size of the convolutional kernel, and "Stride" is the stride size.

```
from keras.layers import Conv2D

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

Four hyperparameters (hyperparameters are the knobs you tune (increase and decrease) when configuring your neural network to improve performance) that control the size and depth of the output volume:

- **Filters:** Convolution filter is also known as a kernel. A filter, or kernel, in a CNN is a small matrix of weights that slides over the input data (such as an image), performs element-wise multiplication with the part of the input it is currently on, and then sums up all the results into a single output pixel. This process is known as convolution.
- **Kernel size:** is the size of the convolutional filter matrix. Sizes vary: 2 × 2, 3 × 3, 5 × 5.

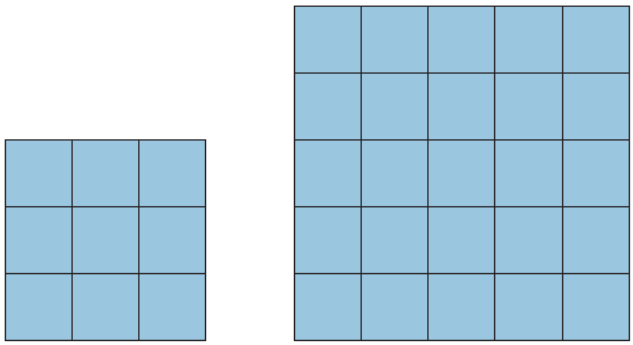


Figure 3.18 The kernel size refers to the dimensions of the convolution filter.

- **Stride:** 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.

Stride Size Definition:

The stride size is the number of pixels the convolutional kernel moves at each step or slide across the input data. A stride of 1 means the kernel moves one pixel at a time, while a larger stride (e.g., 2) means the kernel skips pixels, resulting in a larger step.

Effect on Feature Map Size:

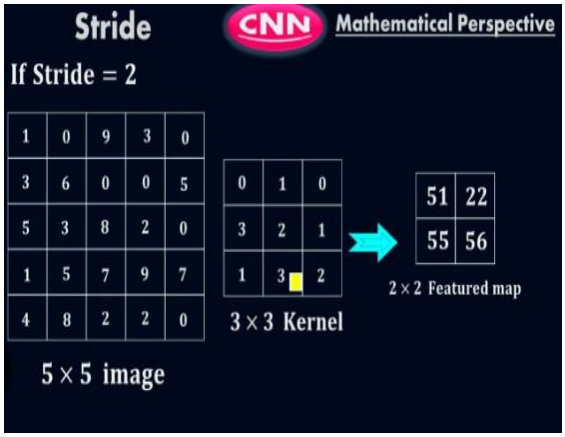
Smaller stride sizes result in larger feature maps because the convolutional kernel overlaps more and covers the input data more comprehensively. Larger stride sizes result in smaller feature maps because the convolutional kernel skips over parts of the input, reducing the coverage.

Mathematical Relationship:

The size of the feature map (output size) can be computed using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data (e.g., width or height), "Kernel size" is the size of the convolutional kernel, and "Stride" is the stride size.



- **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.

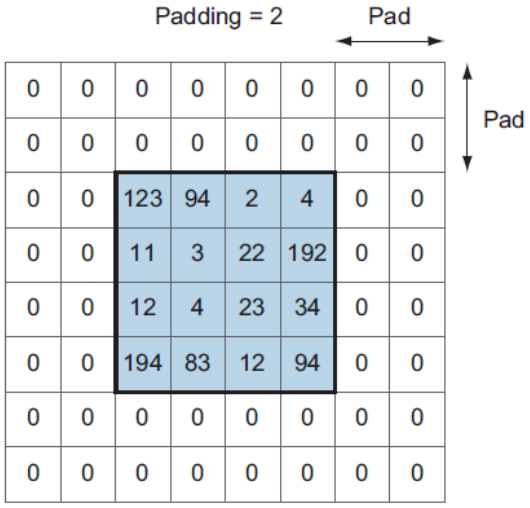


Figure 3.19 Zero-padding adds zeros around the border of the image. Padding = 2 adds two layers of zeros around the border.

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.

Preserving Spatial Dimensions:

- Without Padding: When a convolutional filter is applied to an input, the resulting feature map has smaller dimensions than the original input. For example, if you apply a 3x3 filter to a 5x5 image with a stride of 1, the output will be 3x3, as the filter can't fully cover the edges of the image.
- With Padding: Padding adds extra pixels (typically zeros) around the border of the input image, allowing the filter to process the edges and corners of the image. This preserves the spatial dimensions of the input, ensuring that the output feature map remains the same size as the input.

Mathematical Representation:

The size of the feature map after convolution (or pooling) can be calculated using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data, "Kernel size" is the size of the convolutional kernel, "Padding" is the size of the zero-padding, and "Stride" is the stride size.

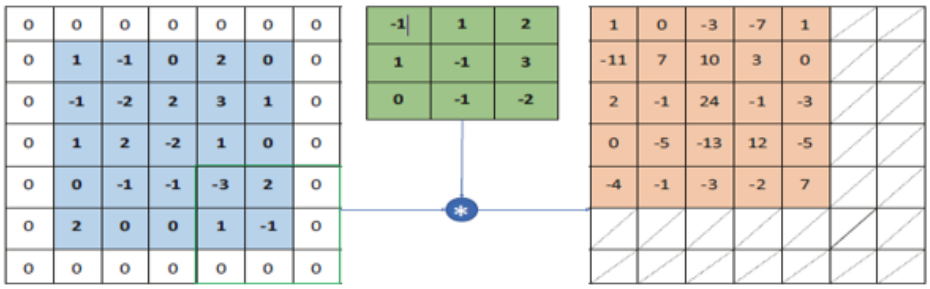
$$h' = \left\lfloor \frac{h + 2 \cdot p_h - k}{s} \right\rfloor + 1$$

$$w' = \left\lfloor \frac{w + 2 \cdot p_w - k}{s} \right\rfloor + 1$$

Taking the preceding example as an example, h = w = 5, k = 3, ph = pw = 1, s = 1, the output are:

$$h' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$

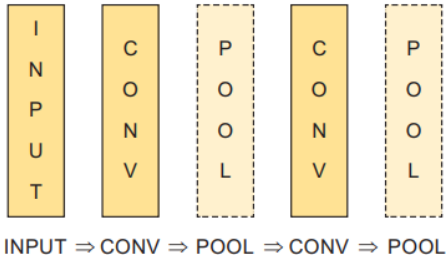
$$w' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$



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 sub sampling and pooling layer is to down sample 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 below).




```
from keras.layers import MaxPooling2D

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

There are three main types of pooling layers:

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.

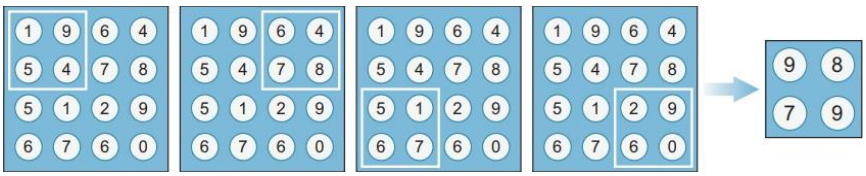


Fig 1.h.pooling filter with a size of 2 × 2 and strides of 2

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 1. i.).

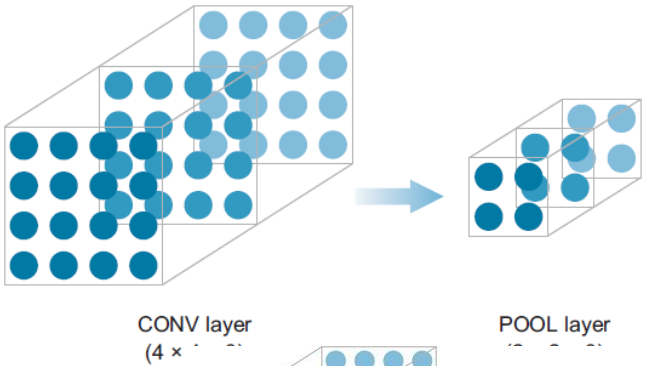
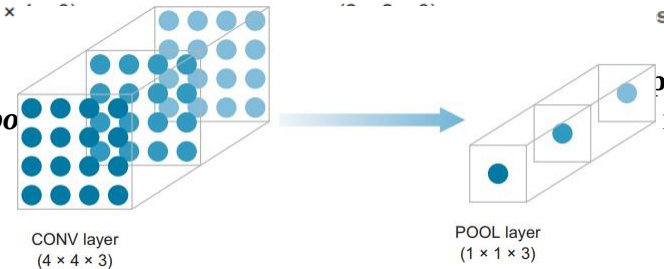


Figure 3.22 If the convolutional layer has three feature maps, the pooling layer's output will have three smaller s.

Global average po **os** reduction. Instead of



setting a window size and strides, global average pooling calculates the values of all pixels in the feature map (figure 1.j).

Figure 1.j The global average pooling layer turns a 3D array into a vector.

Purpose of Pooling Layers:

- Spatial Down-Sampling: Pooling layers reduce the spatial dimensions of the input data, which helps in reducing the computational complexity of the network.Down-sampling also helps in creating a more abstract and hierarchical representation of the input.

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 1.k.).

Fig. 1.k

CONVOLUTIONAL AND POOLING LAYERS SUMMARY:

So far, we have utilized a series of convolutional and pooling layers to process images and extract significant features specific to the training dataset. In summary, here's how we arrived at this point:
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:

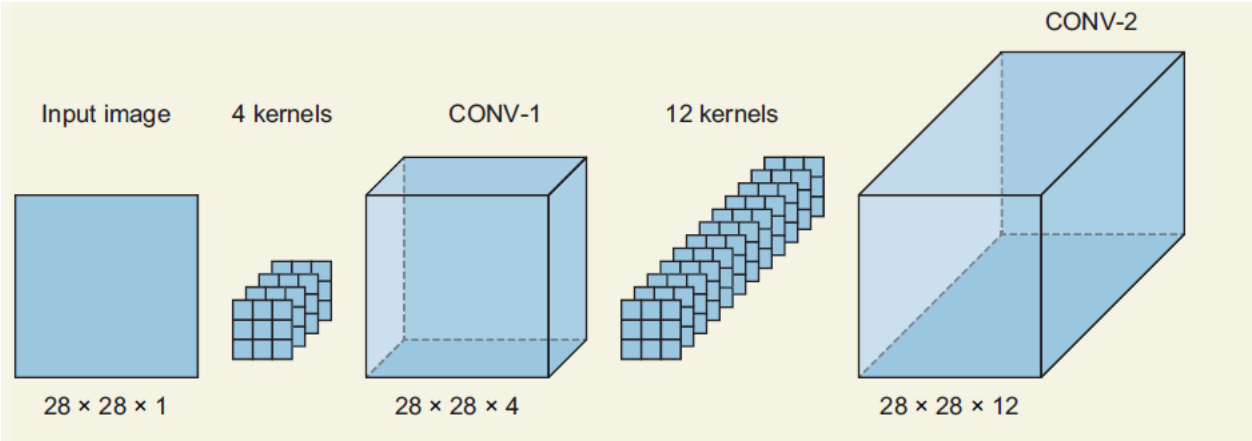
```
from keras.layers import MaxPooling2D

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.

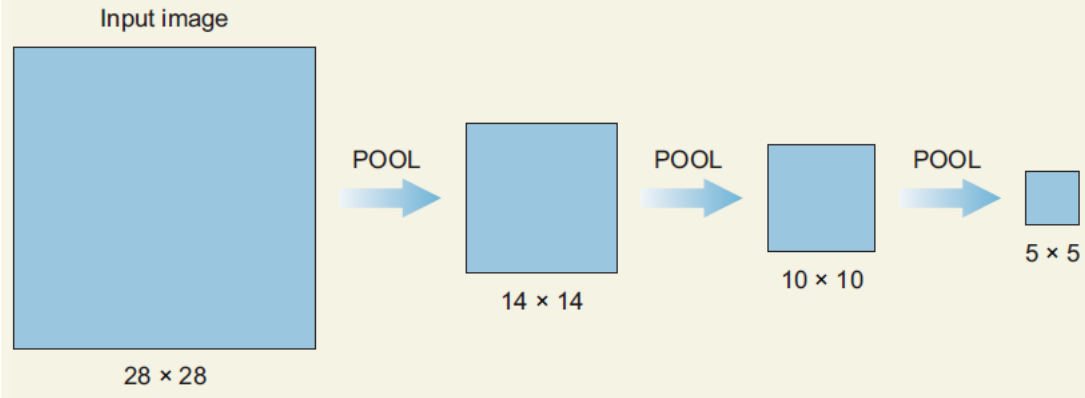
Visualize what happens after each layer:

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-theimage- 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:



3 Fully connected layers:

After passing the image through 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.

Input flattened vector—As illustrated in figure 1.k, for classification feed the all features to flatten it to a vector with the dimensions (1, n). For example, if the features tube has the dimensions of 5 × 5 × 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—The softmax activation function is used 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.

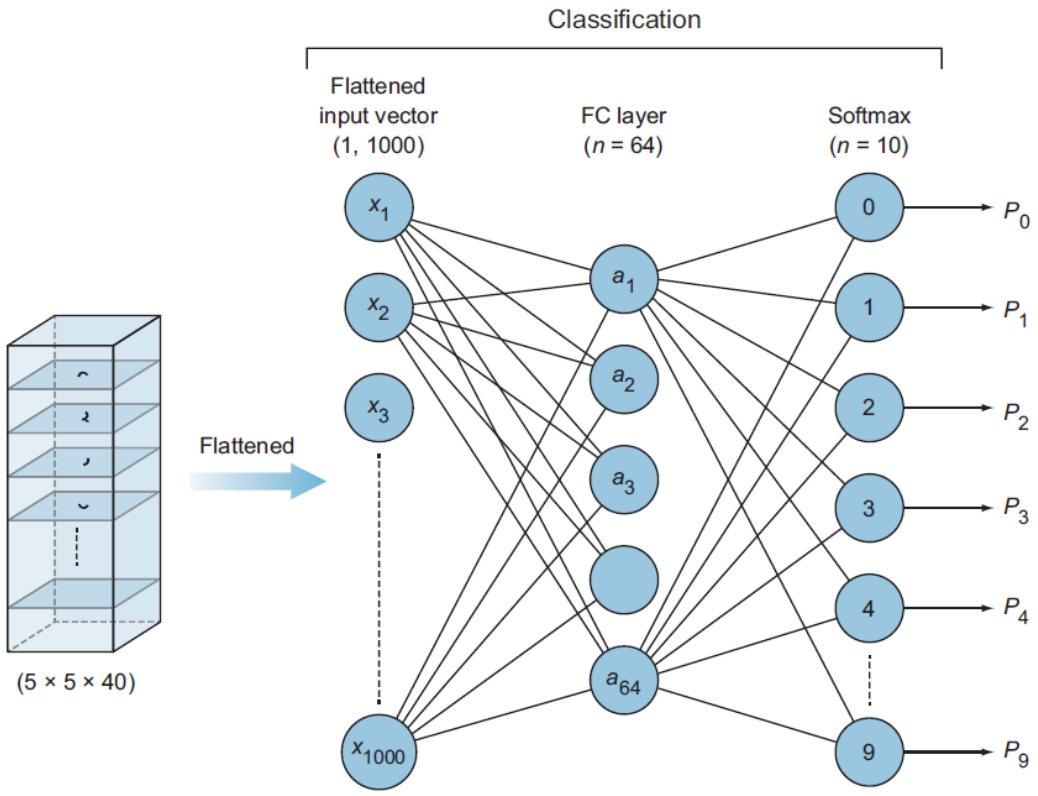


Figure 1.k Fully connected layers for an MLP

Finally, the output layer of a CNN can also be used to perform regularization techniques, such as dropout or batch normalization, to improve the network's performance.

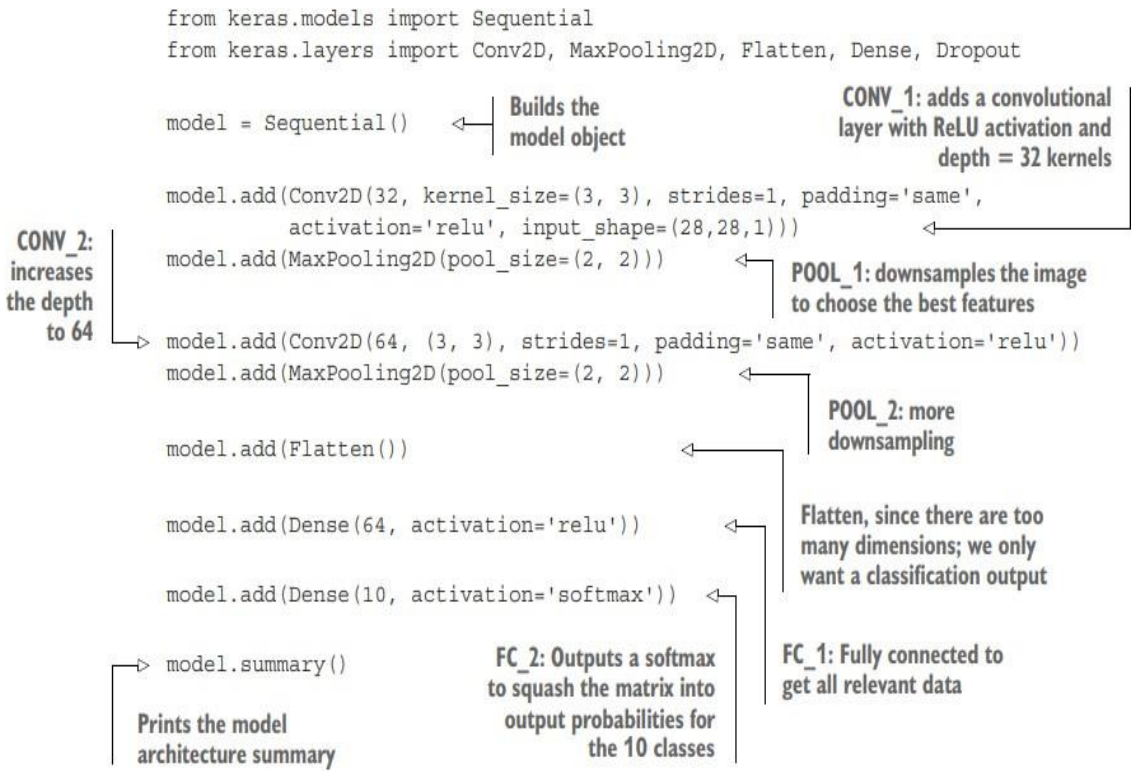
1. Building CNN to Classify 2D Images (Python code)

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.

i) Steps to Classify Images Using CNN:

Data set: the MNIST dataset

Snippet Code:



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

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.

Model Summary Line by Line:

CONV_1—The input shape is (28 × 28 × 1) and 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.

ii) Number of parameters (weights)

Parameters are the weights with which the network learns. The of 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)

to calculate the parameters at the second layer CONV_2

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

the depth of the previous layer is 32, then

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

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

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

The total number of parameters that this network needs to optimize: 220,234.
The Trainable parameters and non-Trainable parameters are:

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

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.

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.

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 occurs when a Convolutional Neural Network (CNN) model fits too well to the training data, making it difficult to generalize to new data. This can happen when the model recognizes specific images instead of general patterns.

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. In overfitting, the network performs very well in the training dataset but performs poorly in the test dataset (figure below)

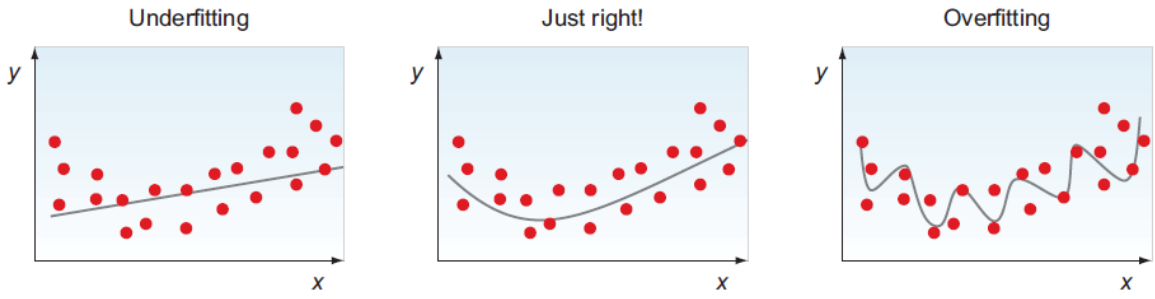


Figure 3.30 Underfitting (left): the model doesn't represent the data very well. Just right (middle): the model fits the data very well. Overfitting (right): the model fits the data too much, so it won't be able to generalize for unseen examples.

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. The word "turns off," means, 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 behaviour 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.

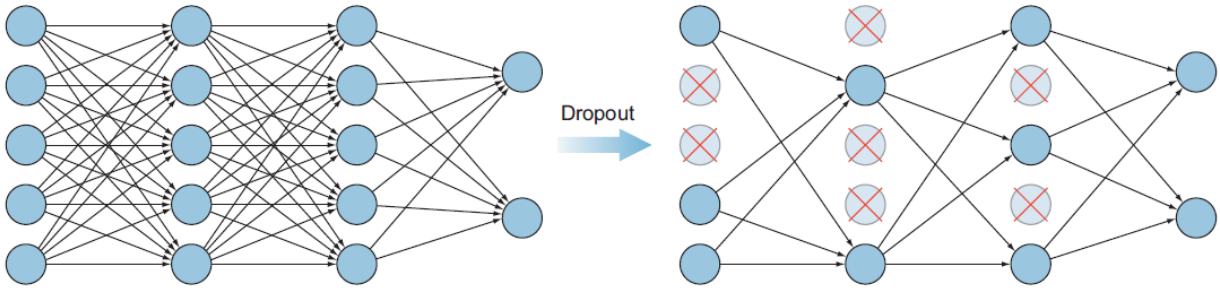


Figure 3.31 Dropout turns off a percentage of the neurons that make up a network layer.

Why do we need dropout layers?

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.

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

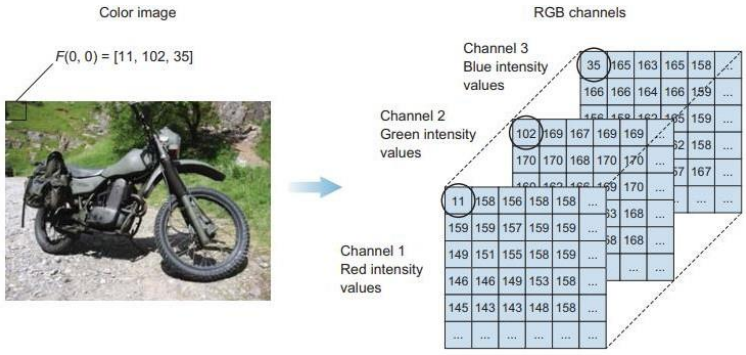
Dropout layer with 50% probability

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

Prints the model architecture summary

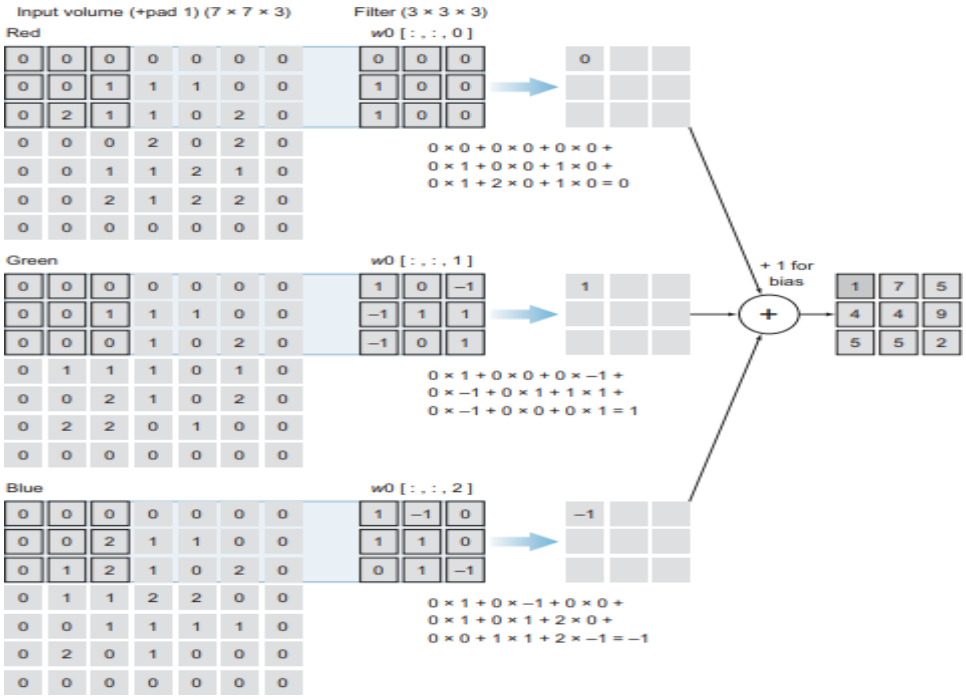
1. Convolution in Colour images (3D images):

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. 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 three matrices represents the value of intensity of its color. When they are stacked, they create a complete color image (figure 1.l).



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.



How do we perform a convolution on a color image?

Sliding the convolutional kernel over the image and compute the feature maps, resulting in a 3D kernel. The kernel is itself three-dimensional: one dimension for each color channel (figure below).

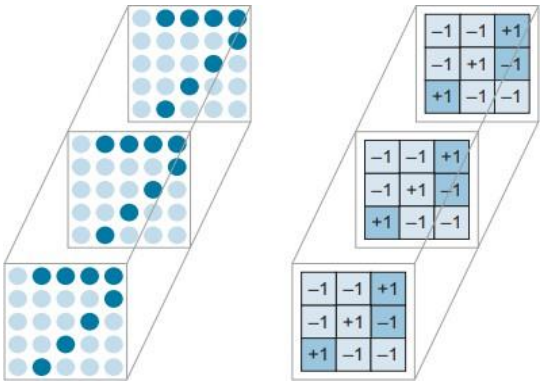


Figure 3.35 Performing convolution in Color images

To perform convolution in color images, sum is three times as many terms. As shown in figure 1.m.

- 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 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.

Problem:

Calculate a value in a Feature Map for 3x3 Image feature convoluted with a 3x3 filter or a Kernel where , Image Feature= [[0,0,0],[0,0,2],[0,1,2]] and kernel =[[1,-1,0],[1,1,0],[0,1,-1]] .

Solution : New Value in a Feature Map = 0*1 +0*-1+0*0+0*1+0*1+2*0+0*0+1*1+2*-1

Answer is -1

In the CNN in figure below, we have an input image of dimensions (7 × 7 × 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.

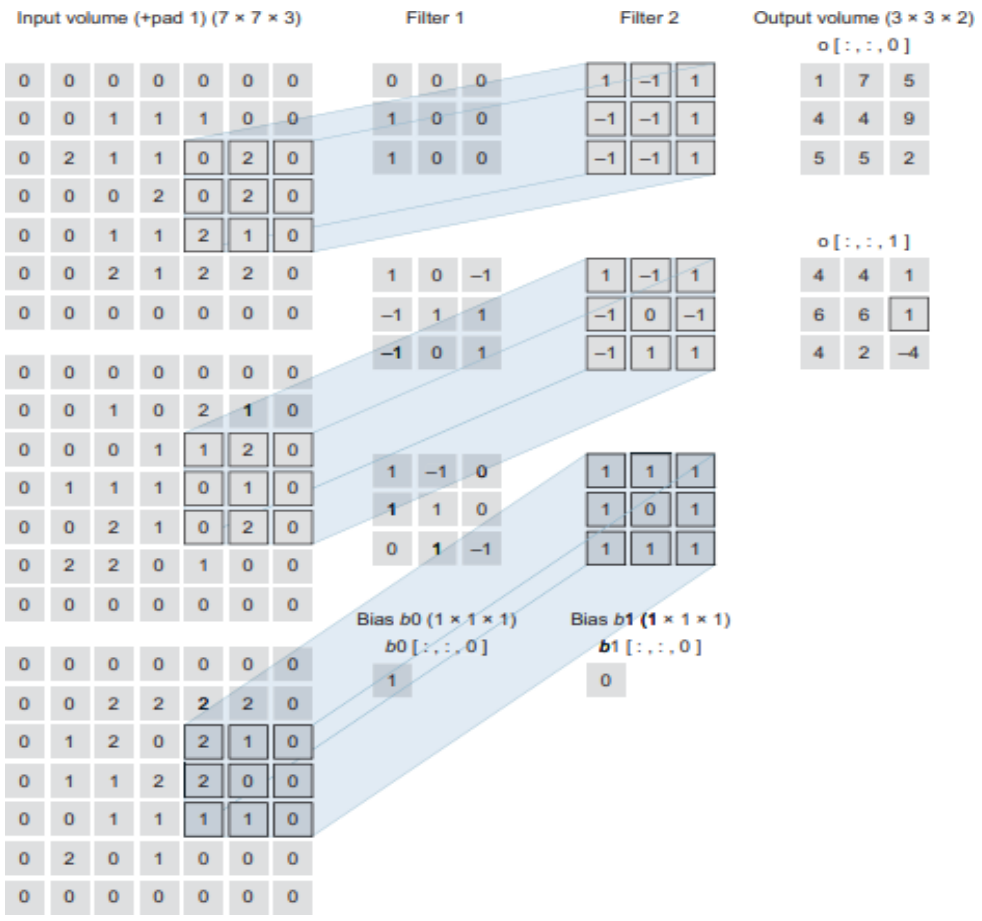


Fig: input image has dimensions (7 × 7 × 3), and we add two convolution filters of dimensions (3 × 3). The output feature map has a depth of 2

Performance Metrics for Classification Models:

Performance metrics allow us to evaluate our model. The simplest way to measure the "goodness" of our model is by measuring its accuracy. To measure Accuracy, precision, recall and F1 score we use Confusion matrix.

Confusion matrix:

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing. Since it shows the errors in the model performance in the form of a matrix, hence also known as an error matrix.

The goal is to describe model performance from different angles other than prediction accuracy. For example, suppose we are building a classifier to predict whether a patient is sick or healthy. The expected classifications are either positive (the patient is sick) or negative (the patient is healthy). We run our model on 1,000 patients and enter the model predictions in Figure below.

	Predicted sick (positive)	Predicted healthy (negative)
Sick patients (positive)	100 True positives (TP)	30 False negative (FN)
Healthy patients (negative)	70 False positives (FP)	800 True negatives (TN)

The most basic terms, which are whole numbers (not rates):

- True positives (TP)—The model correctly predicted yes (the patient has the disease).
- True negatives (TN)—The model correctly predicted no (the patient does not have the disease).
- False positives (FP)—The model falsely predicted yes, but the patient actually does not have the disease (in some literature known as a Type I error or error of the first kind).
- False negatives (FN)—The model falsely predicted no, but the patient actually does have the disease (in some literature known as a Type II error or error of the second kind).

Metrics of Classification with Confusion Matrix:

i)Accuracy: The accuracy metric measures how many times our model made the correct prediction. So, if we test the model with 100 input samples, and it made the correct prediction 90 times, this means the model is 90% accurate.

Here is the equation used to calculate model accuracy:

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total number of examples}}$$

Is accuracy the best metric for evaluating a model?

We have been using accuracy as a metric for evaluating our model in earlier projects, and it works fine in many cases. Consider the following problem: you are designing a medical diagnosis test for a rare disease. Suppose that only one in every million people has this disease. Without any training or even building a system at all, if you hardcode the output to be always negative (no disease found), your system will always achieve 99.999% accuracy. Is that good? The system is 99.999% accurate, which might sound fantastic, but it will never capture the patients with the disease. This means the accuracy metric is not suitable to measure the "goodness" of this model. We need other evaluation metrics that measure different aspects of the model's prediction ability which are precision ,Recall and F1 Score.

Precision and recall

The patients that the model predicts are negative (no disease) are the ones that the model believes are healthy, and we can send them home without further care. The patients that the model predicts are positive (have disease) are the ones that we will send for further investigation. Which mistake would we rather make? Mistakenly diagnosing someone as positive (has disease) and sending them for more investigation is not as bad as mistakenly diagnosing someone as negative (healthy) and sending them home at risk to their life. The obvious choice of evaluation metric here is that we care more about the number of false negatives (FN). We want to find all the sick people, even if the model accidentally classifies some healthy people as sick. This metric is called recall.

Recall (also known as sensitivity) tells us how many of the sick patients our model incorrectly diagnosed as well. In other words, how many times did the model incorrectly diagnose a sick patient as negative (false negative, FN)?

Recall is calculated by the following equation:

$$\text{Recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

Precision (also known as specificity) is the opposite of recall. It tells us how many of the well patients our model incorrectly diagnosed as sick. In other words, how many times did the model incorrectly diagnose a well patient as positive (false positive, FP)?

Precision is calculated by the following equation:

$$\text{Precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

F-score :

In many cases, we want to summarize the performance of a classifier with a single metric that represents both recall and precision. To do so, we can convert precision (p) and recall (r) into a single F-score metric. In mathematics, this is called the harmonic mean of p and r:

$$\text{F-score} = \frac{2pr}{p+r}$$

The F-score gives a good overall representation of how your model is performing.

Let's take a look at the health-diagnostics example again. We agreed that this is a high recall model. But what if the model is doing really well on the FN and giving us a high recall score, but it's performing poorly on the FP and giving us a low precision score? Doing poorly on FP means, in order to not miss any sick patients, it is mistakenly diagnosing a lot of patients as sick, to be on the safe side. So, while recall might be more important for this problem, it is good to look at the model from both scores—precision and recall—together:

	Precision	Recall	F-score
Classifier A	95%	90%	92.4%
Classifier B	98%	85%	91%

Defining the model evaluation metric is a necessary step because it will guide your approach to improving the system. Without clearly defined metrics, it can be difficult to tell whether changes to a DL system result in progress or not.

Problem:

A DL Image classification model is trained to predict tumour in images. The test dataset consists of 100 people. Out which **10 people** who have tumours are predicted positively, **60 people** who don't have tumors are predicted negatively, **22 people** are predicted as positive of having a tumor, although they don't have a tumor and **8 people** who have tumors are predicted as negative. Draw the Confusion Matrix and Find the Accuracy, recall, Precision and F1 Score.

Solution:

Confusion Matrix:

		ACTUAL	
		Negative	Positive
PREDICTION	Negative	60	8
	Positive	22	10

TP= 10 ; TN= 60; FP= 22; FN =8

Accuracy = TP+TN/(TP+TN+FP+FN) = 70 /100 = 0.70 *100**=70%**

Recall(R) = TP/(TP+FN) = 10/18 = 0.5555 *100**= 55.55%**

Precision (P) = TP/(TP+FP) = 10/32 =0.3125*100 **=31.25 %**

F1score=2*P*R/(P+R) = 0.34 /0.86 =0.39* 100 **= 39%**

Designing a baseline model

Now that you have selected the metrics you will use to evaluate your system, it is time to establish a reasonable end-to-end system for training your model. Depending on the problem you are solving, you need to design the baseline to suit your network type and architecture. In this step, you will want to answer questions like these:

Figure below shows the architecture of an AlexNet deep CNN, with the dimensions of each layer. The input layer is followed by five convolutional layers (CONV1 through CONV5), the output of the fifth convolutional layer is fed into two fully connected layers (FC6 through FC7), and the output layer is a fully connected layer (FC8) with a softmax function:

INPUT ⇒ CONV1 ⇒ POOL1 ⇒ CONV2 ⇒ POOL2 ⇒ CONV3 ⇒ CONV4 ⇒ CONV5 ⇒ POOL3 ⇒ FC6 ⇒ FC7 ⇒ SOFTMAX_8

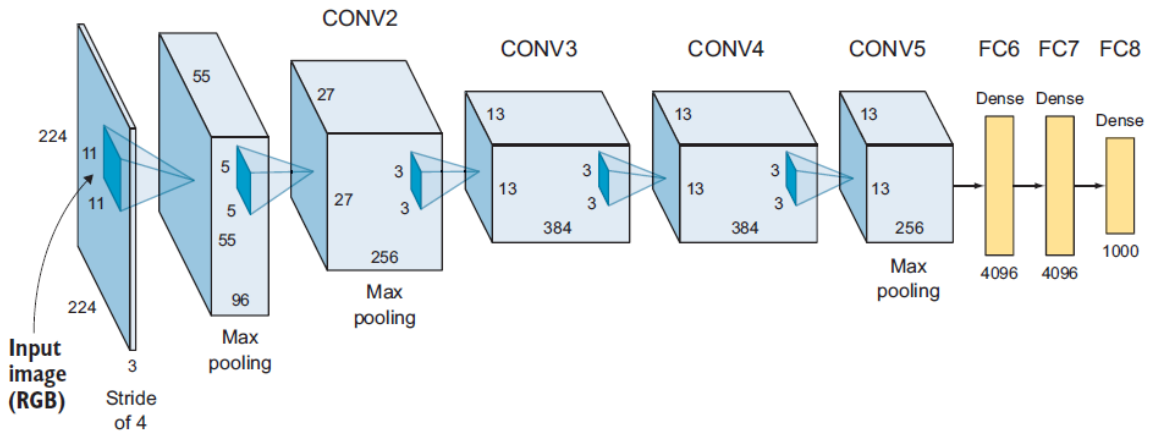


Figure 4.1 The AlexNet architecture consists of five convolutional layers and three FC layers.

Looking at the AlexNet architecture, you will find all the network hyperparameters that you need to get started with your own model:

- Network depth (number of layers): 5 convolutional layers plus 3 fully connected layers
- Layers' depth (number of filters): CONV1 = 96, CONV2 = 256, CONV3 = 384, CONV4 = 385, CONV5 = 256
- Filter size: 11 × 11, 5 × 5, 3 × 3, 3 × 3, 3 × 3
- ReLU as the activation function in the hidden layers (CONV1 all the way to FC7)
- Max pooling layers after CONV1, CONV2, and CONV5
- FC6 and FC7 with 4,096 neurons each
- FC8 with 1000 neurons, using a softmax activation function

1. Getting your data ready for training

- ***Splitting your data for train/validation/test***

When we train a ML model, we split the data into train and test datasets (figure 4.2). We use the training dataset to train the model and update the weights, and then we evaluate the model against the test dataset that it hasn't seen before. The golden rule here is this: never use the test data for training. The reason we should never show the test samples to the model while training is to make sure the model is not cheating. We show the model the training samples to learn their features, and then we test how it generalizes on a dataset that it has never seen, to get an unbiased evaluation of its performance**.**

Figure 4.2 Splitting the data into training and testing datasets

WHAT IS THE VALIDATION DATASET?

After each epoch during the training process, we need to evaluate the model's accuracy and error to see how it is performing and tune its parameters. If we use the test dataset to evaluate the model during training, we will break our golden rule of never using the testing data during training. The test data is only used to evaluate the final performance of the model *after* training is complete. So we make an additional split called a *validation dataset* to evaluate and tune parameters *during* training (figure 4.3). Once the model has completed training, we test its final performance over the test dataset.

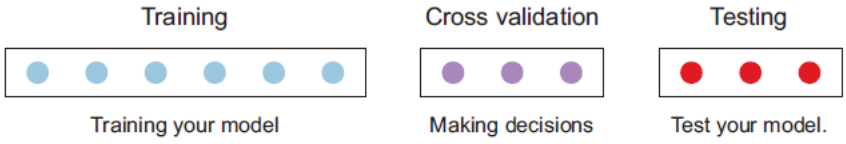


Figure 4.3 An additional split called a *validation dataset* to evaluate the model during training while keeping the test subset for the final test after training

Data preprocessing:

Before you feed your data to the neural network, you will need to do some data cleanup and processing to get it ready for your learning model. There are several preprocessing techniques to choose from, based on the state of your dataset and the problem you are solving. Some of them are:

- **IMAGE GRAYSCALING**
- **IMAGE RESIZING**
- **DATA NORMALIZATION**
- **IMAGE AUGMENTATION**

- **<u>IMAGE GRAYSCALING</u>**

We talked in chapter 3 about how color images are represented in three matrices versus only one matrix for grayscale images; color images add computational complexity with their many parameters. You can make a judgment call about converting all your images to grayscale, if your problem doesn't require color, to save on the computational complexity. A good rule of thumb here is to use the human-level performance rule: if you are able to identify the object with your eyes in grayscale images, then a neural network will probably be able to do the same.

- **IMAGE RESIZING**

One limitation for neural networks is that they require all images to be the same shape. If you are using MLPs, for example, the number of nodes in the input layer must be equal to the number of pixels in the image (remember how, in chapter 3, we flattened the image to feed it to the MLP). The same is true for CNNs. You need to set the input shape of the first convolutional layer. To demonstrate this, let's look at the Keras code to add the first CNN layer:

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

As you can see, we have to define the shape of the image at the first convolutional layer. For example, if we have three images with dimensions of 32 × 32, 28 × 28, and 64 × 64, we have to resize all the images to one size before feeding them to the model.

- **DATA NORMALIZATION**

Data normalization is the process of rescaling your data to ensure that each input feature (pixel, in the image case) has a similar data distribution. Often, raw images are composed of pixels with varying scales (ranges of values). For example, one image may have a pixel value range from 0 to 255, and another may have a range of 20 to 200. Although not required, it is preferred to normalize the pixel values to the range of 0 to 1 to boost learning performance and make the network converge faster.

To make learning faster for your neural network, your data should have the following characteristics:

- *Small values*—Typically, most values should be in the [0, 1] range.
- *Homogenous*—All pixels should have values in the same range.

Data normalization is done by subtracting the mean from each pixel and then dividing the result by the standard deviation. The distribution of such data resembles a Gaussian curve centered at zero. To demonstrate the normalization process, figure 4.5 illustrates the operation in a scatterplot.

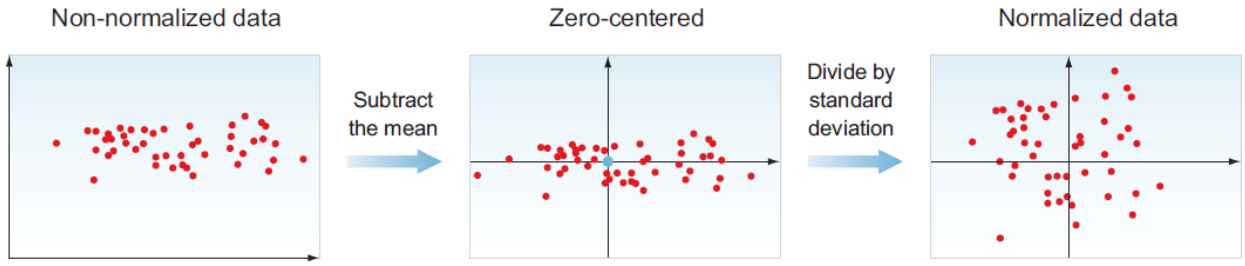


Figure 4.5 To normalize data, we subtract the mean from each pixel and divide the result by the standard deviation.

In non-normalized data, the cost function will likely look like a squished, elongated bowl. After you normalize your features, your cost function will look more symmetric. Figure 4.6 shows the cost function of two features, *F*1 and *F*2.

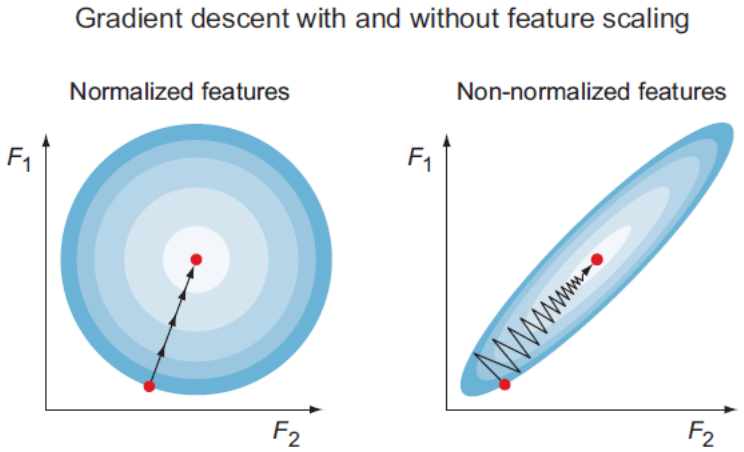


Figure 4.6 Normalized features help the GD algorithm go straight forward toward the minimum error, thereby reaching it quickly (left). With non-normalized features, the GD oscillates toward the direction of the minimum error and reaches the minimum more slowly (right).

As you can see, for normalized features, the GD algorithm goes straight forward toward the minimum error, thereby reaching it quickly. But for non-normalized features, it oscillates toward the direction of the minimum error and ends with a long march down the error mountain. It will eventually reach the minimum, but it will take longer to converge.

- **Image augmentation**

Image augmentation is a technique that creates new images from existing ones by making small changes to them. It's commonly used in computer vision and deep learning to improve the robustness and generalization of machine learning models.

Evaluating the DL model:

a. Interpreting its performance (Underfitted or Overfitted):

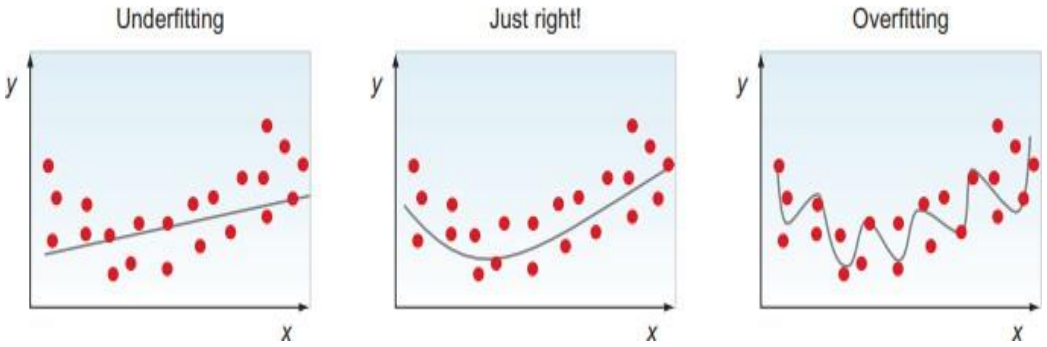
After completing the training of the model, we need to determine the bottlenecks,

- diagnose which CNN components are performing poorly,
- determine whether the poor performance is due to overfitting, underfitting, or a defect in the training data.

The main cause of poor performance in ML and DL models is either overfitting or underfitting on the data.

Underfitting means it fails to learn the training data, so it performs poorly on the training data. This happens when the model is too simple to fit the data or because of more assumptions on data (more Bias) . for example, using one perceptron to classify a nonlinear dataset.

Overfitting, 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 or variance in the Data is more. In overfitting, the network performs very well in the training dataset but performs poorly in the test dataset (figure below)



To diagnose underfitting and overfitting, the two values to focus on while training are the training error and the validation error:

- If the model is doing very well on the training set but relatively poorly on the validation set, then it is overfitting. For example, if train_error is 1% and val_error is 10%, it looks like the model has memorized the training dataset but is failing to generalize on the validation set. In this case, you might consider tuning your hyperparameters to avoid overfitting and iteratively train, test, and evaluate until you achieve an acceptable performance.
- If the model is performing poorly on the training set, then it is underfitting. For example, if the train_error is 14% and val_error is 15%, the model might be too simple and is failing to learn the training set. You might want to consider adding more hidden layers or training longer (more epochs), or try different neural network architectures.

b. Plotting Training and validation loss curves Vs Epochs

Looking at the training verbose output and comparing the error numbers, one way to diagnose overfitting and underfitting is to plot your training and validation errors throughout the training (epochs vs Losses) , as you see in figure below.

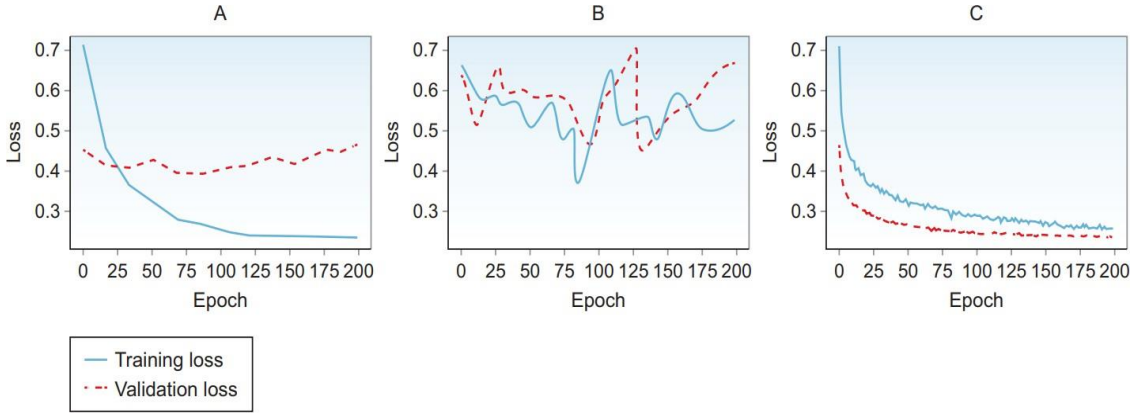


Fig : Evaluating the model with the learning curves

Figure 4.10A shows that the network improves the loss value (aka learns) on the training data but fails to generalize on the validation data. Learning on the validation data progresses in the first couple of epochs and then flattens out and maybe decreases. This is a form of overfitting.

Figure 4.10B shows that the network performs poorly on both training and validation data. In this case, your network is not learning. You don't need more data, because the network is too simple to learn from the data you already have. Your next step is to build a more complex model.

Figure 4.10C shows that the network is doing a good job of learning the training data and generalizing to the validation data. This means there is a good chance that the network will have good performance out in the wild on test data.

2. Improving the NN with Hyperparameter Tuning:

After you run your training experiment and diagnose for overfitting and underfitting, you need to decide whether it is more effective to spend your time tuning the network, cleaning up and processing your data, or collecting more data. The last thing you want to do is to spend a few months working in one direction only to find out that it barely improves network performance. So, before discussing the different hyperparameters to tune, let's answer this question first: should you collect more data?

Collecting more data vs. tuning hyperparameters:

The process involved to make such decisions is:

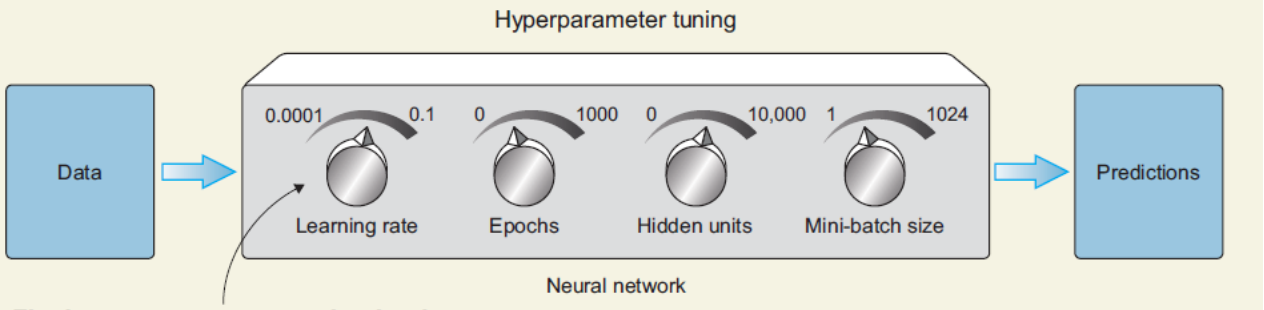
1. Determine whether the performance on the training set is acceptable as-is.
2. Visualize and observe the performance of these two metrics: training accuracy (train_acc) and validation accuracy (val_acc).
3. If the network yields poor performance on the training dataset, this is a sign of underfitting. There is no reason to gather more data, because the learning algorithm is not using the training data that is already available. Instead, try tuning the hyperparameters or cleaning up the training data.
4. If performance on the training set is acceptable but is much worse on the test dataset, then the network is overfitting your training data and failing to generalize to the validation set. In this case, collecting more data could be effective.

Parameters vs. hyperparameters:

Hyperparameters are the variables that we set and tune. Parameters are the variables that the network updates with no direct manipulation from us. Parameters are variables that are learned and updated by the network during training, and we do not adjust them. In neural networks, parameters are the weights and biases that are optimized automatically during the backpropagation process to produce the minimum error. In contrast, hyperparameters are variables that are not learned by the network.

Turning the knobs

Think of hyperparameters as knobs on a closed box (the neural network). Our job is to set and tune the knobs to yield the best performance:



Neural network hyperparameters:

DL algorithms come with several hyperparameters that control many aspects of the model's behavior. Some hyperparameters affect the time and memory cost of running the algorithm, and others affect the model's prediction ability. The challenge with hyperparameter tuning is that there are no magic numbers that work for every problem. This is related to the no free lunch theorem. Good hyperparameter values depend on the dataset and the task at hand. Choosing the best hyperparameters and knowing how to tune them require an understanding of what each hyperparameter does.
Generally speaking, we can categorize neural network hyperparameters into three main categories:

- Network architecture
  - Number of hidden layers (network depth)
  - Number of neurons in each layer (layer width)
  - Activation type
- Learning and optimization
  - Learning rate and decay schedule
  - Mini-batch size
  - Optimization algorithms
  - Number of training iterations or epochs (and early stopping criteria)
- Regularization techniques to avoid overfitting
  - L2 regularization
  - Dropout layers
  - Data augmentation

Tuning each knob up or down and how to know which hyperparameter to tune.

a. Network architecture:

The hyperparameters that define the neural network architecture:

- Number of hidden layers (representing the network depth)
- Number of neurons in each layer, also known as hidden units (representing the network width)
- Activation functions

i) Depth And Width of The Neural Network:

Once the we decide the number of hidden layers in network (depth) and the number of neurons in each layer (width). The number of hidden layers and units describes the learning capacity of the network. The goal is to set the number large enough for the network to learn the data features. A smaller network might underfit, and a larger network might overfit. To know what is a "large enough" network, you pick a starting point, observe the performance, and then tune up or down.

The more complex the dataset, the more learning capacity the model will need to learn its features.

If you provide the model with too much learning capacity (too many hidden units), it might tend to overfit the data and memorize the training set. If your model is overfitting, you might want to decrease the number of hidden units.

Generally, it is good to add hidden neurons until the validation error no longer improves. The trade-off is that it is computationally expensive to train deeper networks. Having a small

number of units may lead to underfitting, while having more units is usually not harmful, with appropriate regularization (like dropout)

ii) Activation functions introduces nonlinearity to our neurons. Without activations, our neurons would pass linear combinations (weighted sums) to each other and not solve any nonlinear problems. The non-linear activation functions like ReLU and its variations (like Leaky ReLU) perform the best in hidden layers. But at output layer using the softmax function solve classification problems.

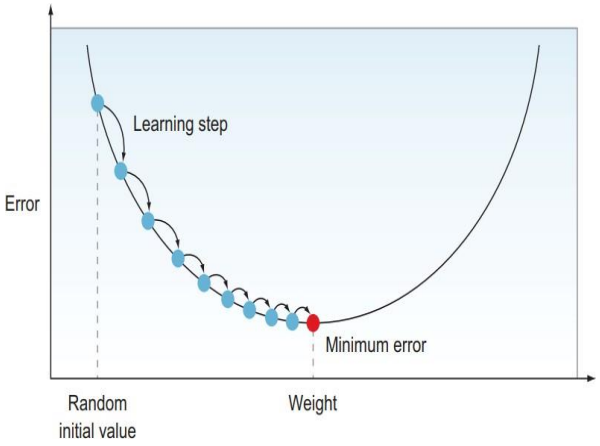
b. Learning and optimization

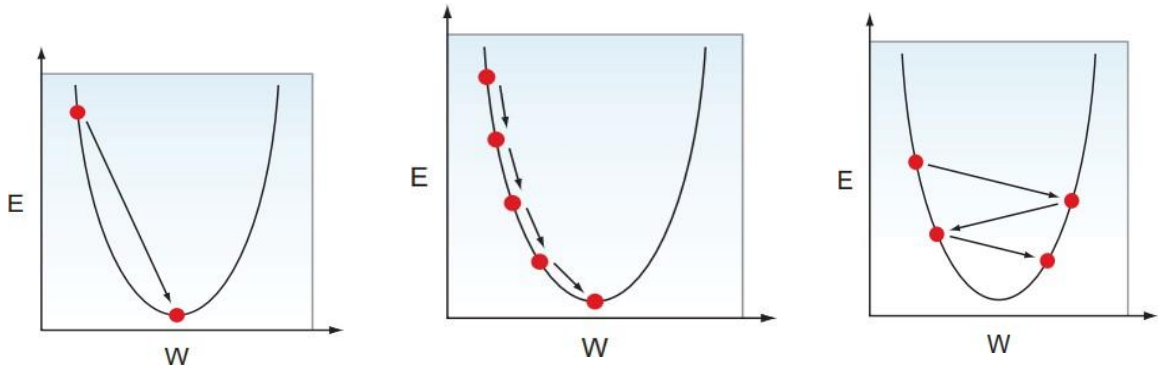
The hyper parameters tuning in an architecture determine how the network learns and optimize its parameter to achieve the minimum error.

i) Learning rate and decay schedule:

The learning rate is the single most important hyperparameter, and one should always make sure that it has been tuned. In GD optimizer searches for the optimal values of weights that yield the lowest error possible. When setting up our optimizer, we need to define the step size

that it takes when it descends the error mountain. This step size is the learning rate. It represents how fast or slow the optimizer descends the error curve. When we plot the cost function with only one weight, we get the oversimplified U-curve in figure 4.14, where the weight is randomly initialized at a point on the curve.





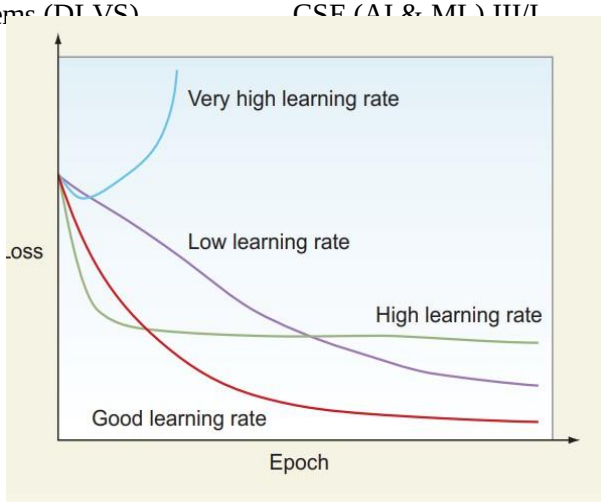
Too-high vs. too-low learning rate Setting the learning rate high or low is a trade-off between the optimizer speed versus performance.

When plotting the loss value against the number of training iterations (epochs), you will notice the following:

- Much smaller lr—The loss keeps

decreasing but needs a lot more time to converge.

- Larger lr—The loss achieves a better value than what we started with, but is still far from optimal.
- Much larger lr—The loss might initially decrease, but it starts to increase as the weight values get farther and farther away from the optimal values.
- Good lr—The loss decreases consistently until it reaches the minimum possible value.



The difference between very high, high, good, and low learning rates are shown in the plot below.

ii)A systematic approach to find the optimal learning rate:

The optimal learning rate will be dependent on the topology of your loss landscape, which in turn is dependent on both your model architecture and your dataset. Whether you are using Keras, Tensorflow, PyTorch, or any other DL library, using the default learning rate value of the optimizer is a good start leading to decent results. Each optimizer type has its own default value. Read the documentation of the DL library that you are using to find out the default value of your optimizer. If your model doesn't train well, you can play around with the lr variable using the usual suspects—0.1, 0.01, 0.001, 0.0001, 0.00001, and 0.000001—to improve performance or speed up training by searching for an optimal learning rate.

The way to debug this is to look at the validation loss values in the training verbose:

- If val_loss decreases after each step, that's good. Keep training until it stops improving.
- If training is complete and val_loss is still decreasing, then maybe the learning rate was so small that it didn't converge yet. In this case, you can do one of two things:

  – Train again with the same learning rate but with more training iterations (epochs) to give the optimizer more time to converge.

  – Increase the lr value a little and train again.
- If val_loss starts to increase or oscillate up and down, then the learning rate is too high and you need to decrease its value.

iii)Learning rate decay and adaptive learning:

By now, it's clear that when we try lower learning values, we have a better chance to get to a lower error point. But training it will take longer. In some cases, training takes so long it becomes infeasible. A good trick is to implement a decay rate in our learning rate. The decay rate tells our network to automatically decrease the lr throughout the training process. For example, we can decrease the lr by a constant value of (x) for each (n) number of steps. This way, we can start with the higher value to take bigger steps toward the minimum, and then gradually decrease the learning rate every (n) epochs to avoid overshooting the ideal lr.

One way to accomplish this is by reducing the learning rate linearly (*linear decay*). For example, you can decrease it by half every five epochs, as shown in figure 4.19.

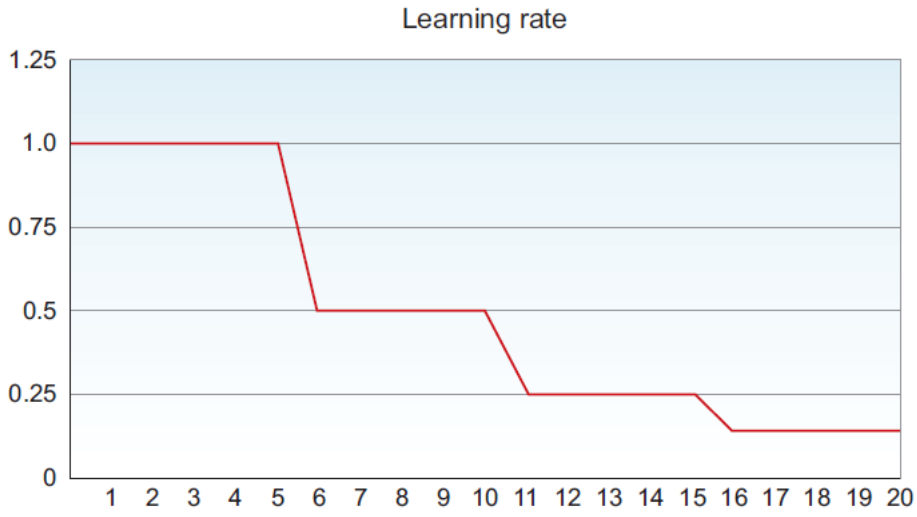


Figure 4.19 Decreasing the lr by half every five epochs

Another way is to decrease the lr exponentially (*exponential decay*). For example, you can multiply it by 0.1 every eight epochs (figure 4.20). Clearly, the network will converge a lot slower than with linear decay, but it will eventually converge.

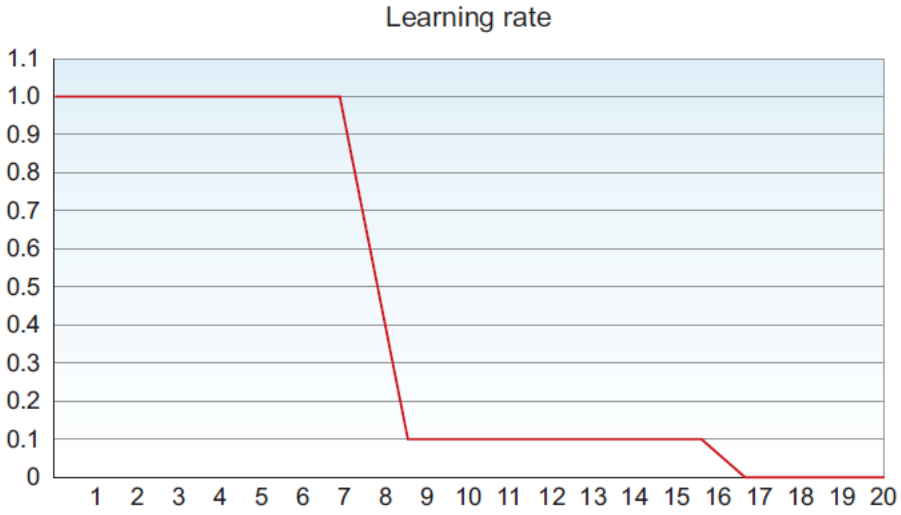
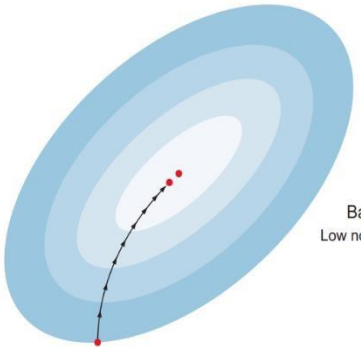


Figure 4.20 Multiplying the lr by 0.1 every eight epochs

iv)Mini-batch size:

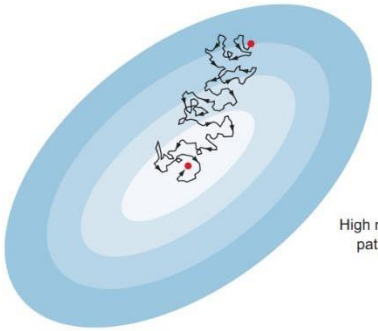
Mini-batch size is another hyperparameter that you need to set and tune in the optimizer algorithm. The batch_size hyperparameter has a big effect on resource requirements of the training process and speed.

*Batch gradient descent (BGD)—T*he entire dataset is fed into the network all at once, apply the feedforward process, calculate the error, calculate the gradient, and backpropagate to update the weights. The optimizer calculates the gradient by looking at the error generated after it sees all the training data, and the weights are updated only once after each epoch. So, in this case, the mini-batch size equals the entire training dataset. The main advantage of BGD is that it has relatively low noise and bigger steps toward the minimum (see figure 4.21). The main disadvantage is that it can take too long to process the entire training dataset at each step, especially when training on big data. BGD also requires a huge amount of memory for training large datasets, which might not be available. BGD might be a good option if you are training on a small dataset.



Batch gradient descent (BGD)
Low noise on its path to the minimum error

Stochastic gradient descent (SGD)—Also called online learning. We feed the network a single instance of the training data at a time and use this one instance to do the forward pass, calculate error, calculate the gradient, and backpropagate to update the weights (figure 4.22). In SGD, the weights are updated after it sees each single instance (as opposed to processing the entire dataset before each step for BGD). SGD can be extremely noisy as it oscillates on its way to the global minimum because it takes a step down after each single instance, which could sometimes be in the wrong direction. This noise can be reduced by using a smaller learning rate, so, on average, it takes you in a good direction and almost always performs better than BGD. With SGD you get to make progress quickly and usually reach very close to the global minimum. The main disadvantage is that by calculating the GD for one instance at a time, you lose the speed gain that comes with matrix multiplication in the training calculations.



Stochastic (GD)
High noise and oscillates on its path to the minimum error

Mini-batch gradient descent (MB-GD)—A compromise between batch and stochastic GD. Instead of computing the gradient from one sample (SGD) or all training samples (BGD), we divide the training sample into mini-batches to compute the gradient from. This way, we can take advantage of matrix multiplication for faster training and start making progress instead of having to wait to train the entire training set.

Optimization algorithms:

The DL community believed that the batch, stochastic, and mini-batch GD algorithms works well, But not. Another by choosing a proper learning rate is challenging because a too small learning rate leads to painfully slow convergence, while a too-large learning rate can hinder convergence and cause the loss function to fluctuate around the minimum or even diverge. We need more creative solutions to further optimize GD.

The most popular gradient-descent-based optimizers are:

- **Momentum**
- **AdaGrad (Adaptive Gradient Algorithm)**
- **RMSProp (Root Mean Square Propagation)**
- **Adam (Adaptive Moment Estimation) and**

SGD is an iterative optimization algorithm employed to minimize a loss function by updating model parameters based on the gradient of the loss. It operates by computing gradients using mini-batches of training data and adjusting parameters in the opposite direction of the gradient to minimize the loss.

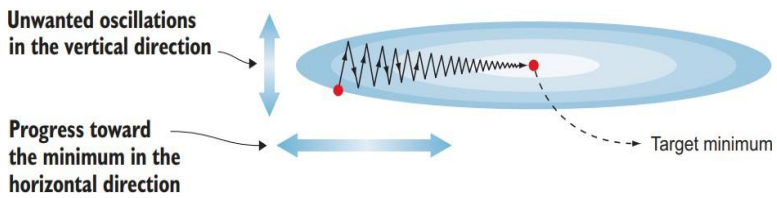
Drawbacks of SGD:

1. Sensitivity to Learning Rate: The choice of learning rate in SGD can be critical, as too large a learning rate may cause the algorithm to overshoot the minimum, while too small a learning rate may result in slow convergence or getting stuck in local minima.

2. Convergence Speed:SGD's reliance on noisy and stochastic gradients often leads to slow convergence, as it frequently oscillates around the optimal solution, taking longer to reach convergence.

3. Inefficiency in High-dimensional Spaces: In high-dimensional optimization problems, SGD may struggle due to the presence of many local minima, making it challenging to find the global minimum efficiently.

SGD ends up with some oscillations in the vertical direction toward the minimum error (figure below). These oscillations slow down the convergence process and make it harder to use larger learning rates, which could result in your algorithm overshooting and diverging.



To reduce these oscillations, a technique called momentum was invented that lets the GD navigate along relevant directions and softens the oscillation in irrelevant directions. In other words, it makes learning slower in the vertical-direction oscillations and faster in the horizontal-direction progress, which will help the optimizer reach the target minimum much faster.

The Exponentially Weighted Moving Average (EWMA):

The Exponentially Weighted Moving Average (EWMA) is commonly used as a smoothing technique in time series. However, due to several computational advantages (fast, low-memory cost), the EWMA is behind the scenes of many optimization algorithms in deep learning, including Gradient Descent with Momentum, RMSprop, Adam, etc.

To understand how the exponential moving average works, let us look at its recursive equation:

$$v_t = \underbrace{\beta v_{t-1}}_{\text{trend}} + \underbrace{(1-\beta)\theta}_{\text{current observation}}$$

Exponential moving average formula

- v_t is a time series that approximates a given variable. Its index t corresponds to the timestamp t. Since this formula is recursive, the value v_0 for the initial timestamp $t = 0$ is needed. In practice, v_0 is usually taken as 0.
- θ is the observation on the current iteration.
- β is a hyperparameter between 0 and 1 which defines how weight importance should be distributed between a previous average value v_{t-1} and the current observation θ

$$v_0 = 0$$

$$\begin{aligned} v_1 &= \beta v_0 + (1-\beta)\theta_1 \\ &= (1-\beta)\cdot\theta_1 \end{aligned}$$

$$\begin{aligned} v_2 &= \beta v_1 + (1-\beta)\theta_2 \\ &= \beta\cdot((1-\beta)\cdot\theta_1) + (1-\beta)\theta_2 \\ &= (1-\beta)\cdot(\theta_2 + \beta\theta_1) \end{aligned}$$

$$\begin{aligned} v_3 &= \beta v_2 + (1-\beta)\theta_3 \\ &= \beta\cdot((1-\beta)\cdot(\theta_2 + \beta\theta_1)) + (1-\beta)\theta_3 \\ &= (1-\beta)\cdot(\theta_3 + \beta\theta_2 + \beta^2\theta_1) \end{aligned}$$

$$\begin{aligned} v_4 &= \beta v_3 + (1-\beta)\theta_4 \\ &= \beta\cdot((1-\beta)\cdot(\theta_3 + \beta\theta_2 + \beta^2\theta_1)) + (1-\beta)\theta_4 \\ &= (1-\beta)\cdot(\theta_4 + \beta\theta_3 + \beta^2\theta_2 + \beta^3\theta_1) \end{aligned}$$

$$\cdots$$

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta)\theta_t \\ &= \beta\cdot((1-\beta)\cdot(\theta_{t-1} + \beta\theta_{t-2} + \ldots + \beta^{t-4}\theta_3 + \beta^{t-3}\theta_2 + \beta^{t-2}\theta_1)) + (1-\beta)\theta_t \\ &= (1-\beta)\cdot(\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \ldots + \beta^{t-3}\theta_3 + \beta^{t-2}\theta_2 + \beta^{t-1}\theta_1) \end{aligned}$$

Obtaining formula for the t-th timestamp

We can see that the most recent observation θ has a weight of 1, the second last observation — β, the third last — β^2, etc. Since $0 < \beta < 1$, the multiplication term β^k goes exponentially down

with the increase of k, *so the older the observations, the less important they are*. Finally, every sum term is multiplied by (1 —β).
In practice, the value for β is usually chosen close to 0.9.
Now we will use this EWMA to implement Momemtum.

<u>Gradient descent with momentum</u>

The momentum is built by adding a velocity term to the equation that updates the weight:

$$w_{new} = w_{old} - \alpha \frac{dE}{dw_i}$$ ← Original update rule

$w_{new} = w_{old}$ – learning rate × gradient + velocity term ← New rule after adding velocity

Momentum or SGD with momentum is a method which helps accelerate gradients vectors in the right directions, thus leading to faster converging.

Gradient Descent with Momentum

initialize $V_{dW} = 0, V_{db} = 0$
for t iterations :

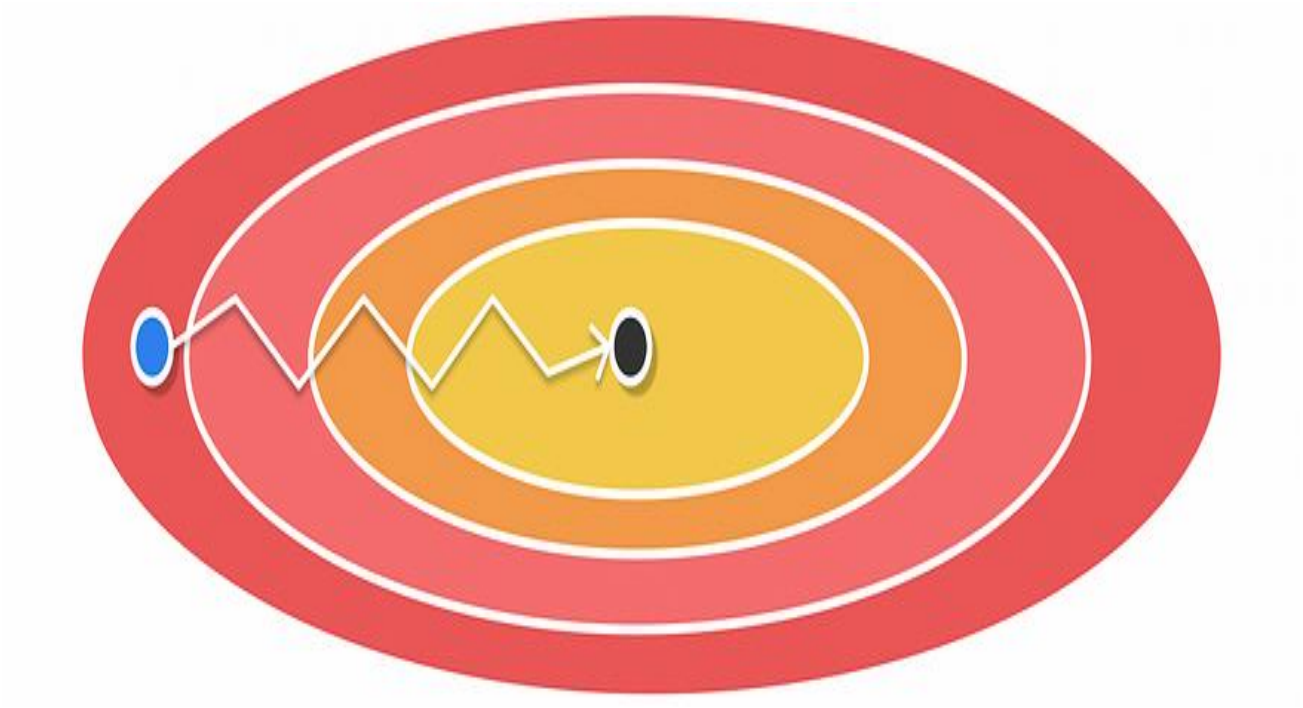
Compute dW, db for current mini batch

$V_{dW} = \beta V_{dW} + (1 - \beta) dW$

$V_{db} = \beta V_{db} + (1 - \beta) db$

$W = W - \alpha V_{dW}$

$b = b - \alpha V_{db}$



Momentum is a technique used in Stochastic Gradient Descent to accelerate the convergence of the optimization process. It involves adding a fraction of the previous update to the current update of the model parameters.
Momentum adds onto gradient descent by considering previous gradients (the slope of the hill *prior* to where the ball is *currently* at). So in the previous case, instead of stopping when the gradient is 0 at the first local minimum, momentum will continue to move the ball forward because it takes into consideration how steep the slope before it was.

Even though momentum with gradient descent converges better and faster, it still doesn't resolve all the problems. First, the hyperparameter η (learning rate) has to be tuned manually. Second, in some cases, where, even if the learning rate is low, the momentum term and the current gradient can alone drive and cause oscillations.

Disadvantages of Momentum:

- The momentum term can cause the optimization process to oscillate around the global minimum i.e. Problem with momentum is oscillations.
- If the momentum parameter β is too high, the optimizer can accumulate momentum from previous updates, causing it to overshoot the optimal solution.
- This leads to the parameter updates oscillating back and forth around the minimum instead of converging smoothly.

First, the Learning rate problem can be further resolved by using other variations of Gradient Descent like AdaptiveGradient and RMSprop.

2. Adagrad:

One of the earliest adaptive learning rate algorithms is AdaGrad, which Duchi et al. introduced in 2011. Its main objective is to hasten convergence for sparse gradient parameters. Each parameter's previous gradient information is tracked by the algorithm, which then modifies the learning rate as necessary.

The Need for Adaptive Learning Rates

Gradient Descent and other conventional optimization techniques use a fixed learning rate throughout the duration of training. However, this uniform learning rate might not be the best option for all

parameters, which could cause problems with convergence. Some parameters might need more frequent updates to hasten convergence, while others might need smaller changes to prevent overshooting the ideal value.

In SGD and mini-batch SGD, the value of η used to be the same for each weight, or say for each parameter. Typically, η = 0.01.

But in Adagrad Optimizer the core idea is that each weight has a different learning rate (η).

The weight updating formula for adagrad looks like:

$$w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w(t-1)}$$

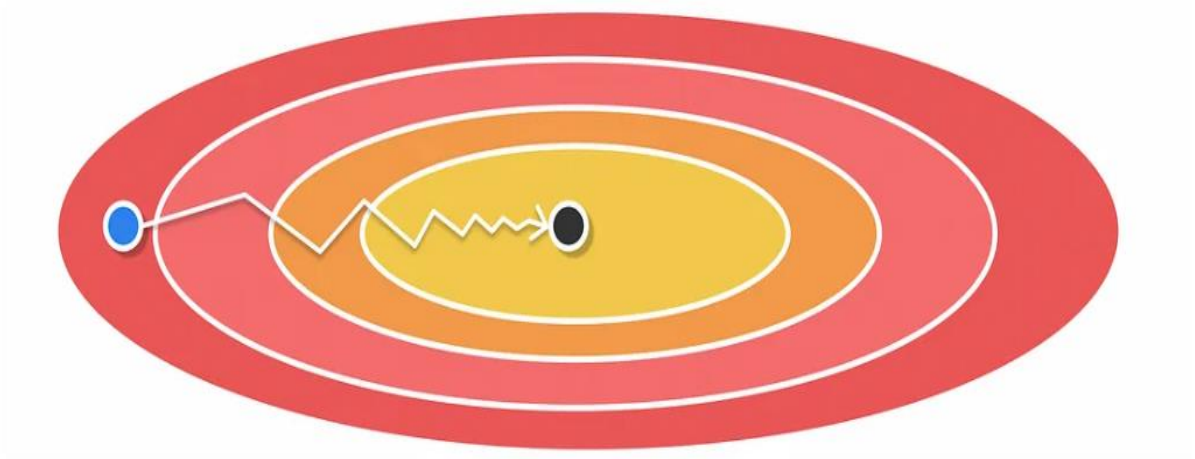
Where

$$\eta'_t = \frac{\eta}{\text{sqrt}(\alpha_t + \epsilon)}$$

epsilon is a small positive value number to avoid divide by zero error

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial \, loss}{\partial w_i} \right)^2$$

The learning rate constantly decays with the increase of iterations (the learning rate is always divided by a positive cumulative number). Therefore, the algorithm tends to converge slowly during the last iterations where it becomes very low.



Optimization with AdaGrad

- The greatest advantage of AdaGrad is that there is no longer a need to manually adjust the learning rate as it adapts itself during training.
- Nevertheless, there is a negative side of AdaGrad: the learning rate constantly decays with the increase of iterations (the learning rate is always divided by a positive cumulative number). Therefore, the algorithm tends to converge slowly during the last iterations where it becomes

very low.

Disadvantages:

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial \text{loss}}{\partial w_i} \right)^2$$

Learning rate at time t,

$$\eta_t = \frac{\eta_{t-1}}{\sqrt{\alpha_t + \epsilon}}$$

Then, here the denominator will become high, which means the entire term will become very low. Which means that the step size will be very low.

Now, while finding the new weight using

$$w_t = w_{t-1} - \eta \left(\frac{\partial \text{loss}}{\partial . w .} \right)_{t-1}$$

The current weight at time 't' and previous weight at time 't-1' becomes almost same. But we don't want this , we want weights to keep optimizing .

The solution to this is RMSprop.

Regularization techniques to avoid overfitting

When neural network is overfitting on the training data, we should optimize and avoid overfit.

Regularization is a technique used to address overfitting by directly changing the architecture of the model by modifying the model's training process. The following are the commonly used regularization techniques:

a. **L1 and L2 regularization**

b. **Dropout layers**

c. **Data augmentation**

d. **Batch Normalization**

L1 and L2 regularization are methods used to mitigate overfitting in machine learning models. These techniques are often applied when a model's data set has a large number of features, and a less complex model is needed.
A regression model that uses the L1 regularization technique is called lasso regression, and a model that uses the L2 is called ridge regression.

L1 vs. L2 Regularization Methods:

- **L1 Regularization:** Also called a lasso regression, adds the absolute value of the sum ("absolute value of magnitude") of coefficients as a penalty term to the loss function.
- **L2 Regularization:** Also called a ridge regression, adds the squared sum ("squared magnitude") of coefficients as the penalty term to the loss function.

i). L1 regularization:

Lasso Regression

- A regression model which uses the **L1 Regularization** technique is called **LASSO(Least Absolute Shrinkage and Selection Operator)** regression. **Lasso Regression** adds the *"absolute value of magnitude"* of the coefficient as a penalty term to the loss function(L). Lasso regression also helps us achieve feature selection by penalizing the weights to approximately equal to zero if that feature does not serve any purpose in the model.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} |w_i|$$

where,

- ***m*** *– Number of Features*
- ***n*** *– Number of Examples*
- ***y_i*** *– Actual Target Value*
- ***y_i(hat)*** *– Predicted Target Value*

ii) L2 regularization:

Ridge Regression

A regression model that uses the **L2 regularization** technique is called **Ridge regression**. **Ridge regression** adds the "*squared magnitude*" of the coefficient as a penalty term to the loss function(L).

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} w_i^2$$

According to regression analysis, L2 regularization is also called ridge regression. In this type of regularization, the squared magnitude of the coefficients or weights multiplied with a regularizer term is added to the loss or cost function. L2 regression can be represented with the following mathematical equation.

The basic idea of L2 regularization is that it penalizes the error function by adding a regularization term to it. This, in turn, reduces the weight values of the hidden units and makes them too small, very close to zero, to help simplify the model.

we update the error function by adding the regularization term:

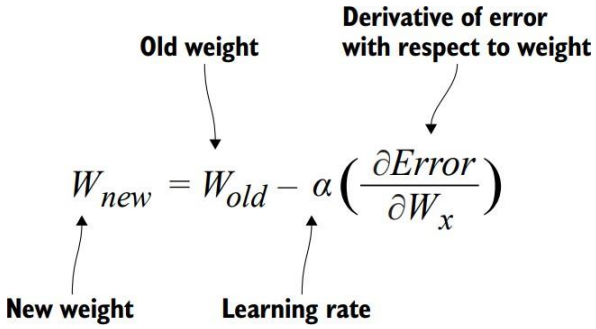
$$\text{error function}_{\text{new}} = \text{error function}_{\text{old}} + \text{regularization term}$$

$$\text{L2 regularization term} = \frac{\lambda}{2m} \times \sum \|w\|^2$$

where lambda (λ) is the regularization parameter, m is the number of instances, and w is the weight. The updated error function looks like this:

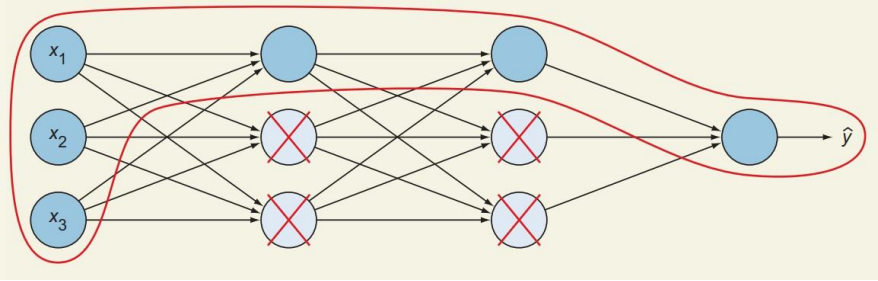
$$\text{error function}_{\text{new}} = \text{error function}_{\text{old}} + \frac{\lambda}{2m} \times \sum \|w\|^2$$

Why does L2 regularization reduce overfitting? Well, we saw how the weights are updated during the backpropagation process. The optimizer calculates the derivative of the error, multiplies it by the learning rate, and subtracts this value from the old weight. Here is the backpropagation equation that updates the weights:



Since we add the regularization term to the error function, the new error becomes larger than the old error. This means its derivative (∂Error/∂Wx) is also bigger, leading to a smaller Wnew. L2 regularization is also known as weight decay, as it forces the weights to decay toward zero (but not exactly zero).

L2 regularization does not make the weights equal to zero. It just makes them smaller to reduce their effect. A large regularization parameter (λ) lead to negligible weights. When the weights are negligible, the model will not learn much from these units. This will make the network simpler and thus reduce overfitting.



L2 regularization reduces the weights and simplifies the network to reduce overfitting.

```
model.add(Dense(units=16, kernel_regularizer=regularizers.l2(λ),
    activation='relu'))
```

When adding a hidden layer to your network, add the kernel_regularization argument with the L2 regularizer

i) Dropout layers:

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). Dropout is another regularization technique that is very effective for simplifying a neural network and avoiding overfitting.

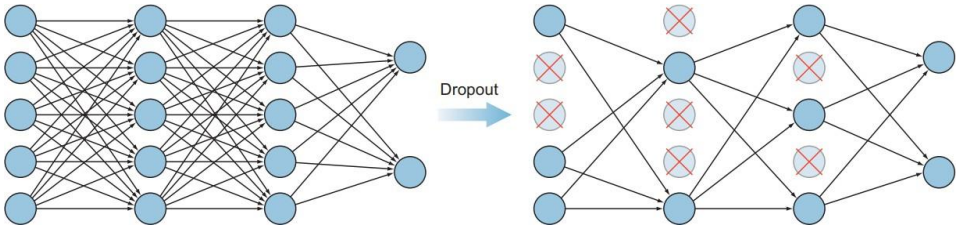


Figure 3.31 Dropout turns off a percentage of the neurons that make up a network layer.

Both L2 regularization and dropout aim to reduce network complexity by reducing its neurons' effectiveness. The difference is that dropout completely cancels the effect of some neurons with every iteration, while L2 regularization just reduces the weight values to reduce the neurons' effectiveness. Both lead to a more robust, resilient neural network and reduce overfitting. It is recommended that you use both types of regularization techniques in your network.

ii) Data augmentation:

One way to avoid overfitting is to obtain more data. Since this is not always a feasible option, we can augment our training data by generating new instances of the same images with some transformations. Data augmentation can be an inexpensive way to give your learning algorithm more training data and therefore reduce overfitting. The many image-augmentation techniques include flipping, rotation, scaling, zooming, lighting conditions, and many other transformations that you can apply to your dataset to provide a variety of images to train on.

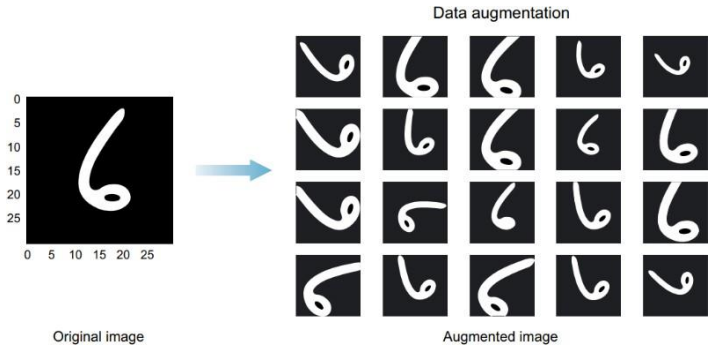
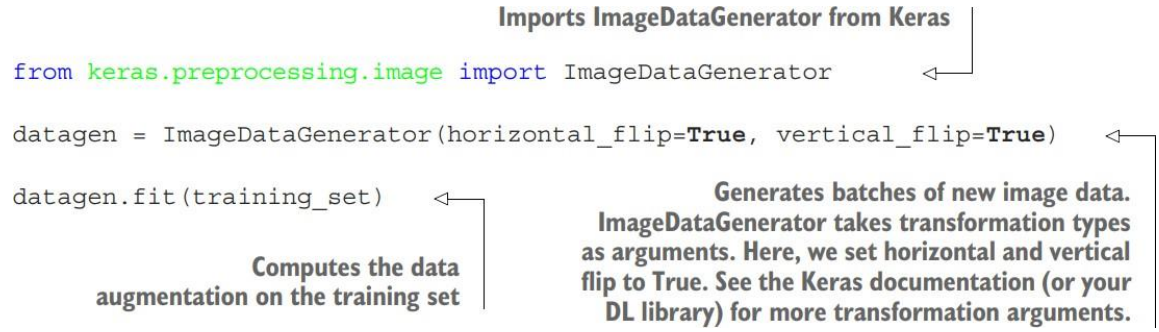


Figure : Various image augmentation techniques applied to an image of the digit 6

In figure above, we created 20 new images that the network can learn from. The main advantage of synthesizing images like this is that now you have more data (20×) that tells your algorithm that if an image is the digit 6, then even if you flip it vertically or horizontally or rotate it, it's still the digit 6. This makes the model more robust to detect the number 6 in any form and shape.

Data augmentation is considered a regularization technique because allowing the network to see many variants of the object reduces its dependence on the original form of the object during feature learning. This makes the network more resilient when tested on new data.

Data augmentation in Keras:



What is Batch Normalization?

Batch normalization is a deep learning approach that has been shown to significantly improve the efficiency and reliability of neural network models. It is particularly useful for training very deep networks, as it can help to reduce the internal covariate shift that can occur during training.

Batch normalization is a supervised learning method for normalizing the interlayer outputs of a neural network. As a result, the next layer receives a "reset" of the output distribution from the preceding layer, allowing it to analyze the data more effectively.

The term "internal covariate shift" is used to describe the effect that updating the parameters of the layers above it has on the distribution of inputs to the current layer during deep learning training. This can make the optimization process more difficult and can slow down the convergence of the model.

Since normalization guarantees that no activation value is too high or too low, and since it enables each layer to learn independently from the others, this strategy leads to quicker learning rates.

By standardizing inputs, the "dropout" rate (the amount of information lost between processing stages) may be decreased. That ultimately leads to a vast increase in precision across the board.

How does batch normalization work?

Batch normalization is a technique used to improve the performance of a deep learning network by first removing the batch mean and then splitting it by the batch standard deviation.

Stochastic gradient descent is used to rectify this standardization if the loss function is too big, by shifting or scaling the outputs by a parameter, which in turn affects the accuracy of the weights in the following layer.

When applied to a layer, batch normalization multiplies its output by a standard deviation parameter (gamma) and adds a mean parameter (beta) to it as a secondary trainable parameter. Data may be "denormalized" by adjusting just these two weights for each output, thanks to the synergy between batch normalization and gradient descents. Reduced data loss and improved network stability were the results of adjusting the other relevant weights.

The goal of batch normalization is to stabilize the training process and improve the generalization ability of the model. It can also help to reduce the need for careful initialization of the model's weights and can allow the use of higher learning rates, which can speed up the training process.

Batch normalization overfitting

While batch normalization can help to reduce overfitting, it is not a guarantee that a model will not overfit. Overfitting can still occur if the model is too complex for the amount of training data, if there is a lot of noise in the data, or if there are other issues with the training process. It is important to use other regularization techniques like dropout, and to monitor the performance of the model on a validation set during training to ensure that it is not overfitting.

Batch normalization equations

During training, the activations of a layer are normalized for each mini-batch of data using the following equations:

To zero-center the inputs, the algorithm needs to calculate the input mean and standard deviation (the input here means the current mini-batch: hence the term batch normalization):

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad \longleftarrow \text{Mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \longleftarrow \text{Mini-batch variance}$$

where m is the number of instances in the mini-batch, μB is the mean, and σB is the standard deviation over the current mini-batch. 2 Normalize the input:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

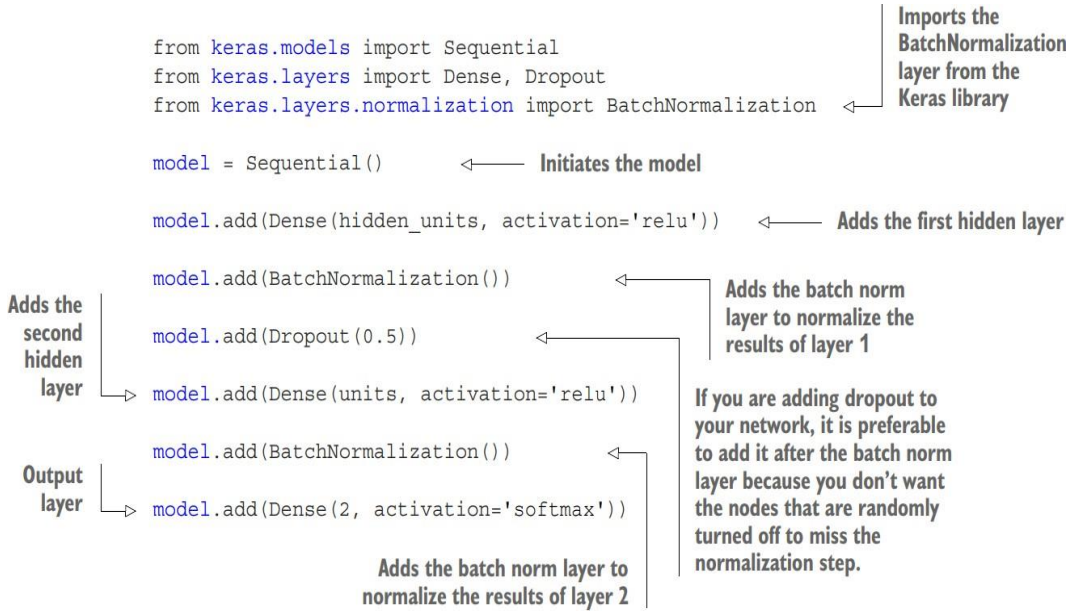
where xˆ is the zero-centered and normalized input. Note that there is a variable here that we added (ε). This is a tiny number (typically 10–5) to avoid division by zero if σ is zero in some estimates. 3 Scale and shift the results. We multiply the normalized output by a variable γ to scale it and add (β) to shift it

$$y_i \leftarrow \gamma X_i + \beta$$

where yi is the output of the BN operation, scaled and shifted.

BN introduces two new learnable parameters to the network: γ and β. So our optimization algorithm will update the parameters of γ and β just like it updates weights and biases. In practice, this means you may find that training is rather slow at first, while GD is searching for the optimal scales and offsets for each layer, but it accelerates once it’s found reasonably good values.

The following code snippet shows you how to add a BN layer when building your neural network:


from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers.normalization import BatchNormalization
Imports the BatchNormalization layer from the Keras library
model = Sequential()
Initiates the model
model.add(Dense(hidden_units, activation='relu'))
Adds the first hidden layer
model.add(BatchNormalization())
Adds the batch norm layer to normalize the results of layer 1
Adds the second hidden layer
model.add(Dropout(0.5))
If you are adding dropout to your network, it is preferable to add it after the batch norm layer because you don't want the nodes that are randomly turned off to miss the normalization step.
model.add(Dense(units, activation='relu'))
model.add(BatchNormalization())
Output layer
model.add(Dense(2, activation='softmax'))
Adds the batch norm layer to normalize the results of layer 2

3. RMSProp

RMSProp addresses issues of diminishing learning rates by adapting them based on the magnitude of recent gradients, making it effective for optimizing neural network training across various architectures and datasets. It adapts the learning rates dynamically, scaling them inversely proportional to the square root of the accumulated squared gradients . This adaptive adjustment helps to stabilize the learning process and accelerate convergence, particularly in deep neural networks with complex and varying gradients.

RMSProp is an extension of gradient descent that uses a moving average of squared gradients to scale the learning rate for each parameter. This helps stabilize the learning process and prevent oscillations in the optimization trajectory.

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db$$

$$W = W - \alpha \cdot v_{dw}$$

$$b = b - \alpha \cdot v_{db}$$

Gradient descent with momenttum

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

RMSprop optimizer

Advantages of RMSProp:

RMSProp offers several advantages over standard SGD:

- Adaptive Learning Rates: By adjusting the learning rate for each parameter, RMSProp can handle different scales of data and varying curvatures of loss functions.
- Convergence Speed: RMSProp can converge faster than SGD with momentum, especially in scenarios with noisy or sparse gradients.
- Stability: The method avoids the diminishing learning rates found in Adagrad, which can stall the

training process in the later stages.

4. Adam:

- Adam is one of the most powerful optimization algorithm used for training deep neural networks.
- It combines the benefits of momentum-based methods (smooth gradient updates) and adaptive learning rate methods (individual learning rates for each parameter).
- Adam often converges faster than other gradient descent algorithms due to its adaptive nature and the combination of momentum and RMSProp methods.

Adam stands for adaptive moment estimation. Adam keeps an exponentially decaying average of past gradients, similar to momentum. Whereas momentum can be seen as a ball rolling down a slope, Adam behaves like a heavy ball with friction to slow down the momentum and control it. Adam usually outperforms other optimizers because it helps train a neural network model much more quickly than the other techniques.

Momentum

$$V_{dW} = \beta_1 V_{dW_{prev}} + (1-\beta) dW$$

$$V_{dB} = \beta_1 V_{dB_{prev}} + (1-\beta) \cdot dB$$

$$W = W - \alpha \cdot V_{dW}$$

$$B = B - \alpha \cdot V_{dB}$$

RMSprop

$$S_{dW} = \beta \cdot S_{dW_{prev}} + (1-\beta)(dW)^2$$

$$S_{dB} = \beta \cdot S_{dB_{prev}} + (1-\beta)(dB)^2$$

$$W = W - \alpha \cdot (dW / \sqrt{S_{dW} + \varepsilon})$$

$$B = B - \alpha \cdot (dB / \sqrt{S_{dB} + \varepsilon})$$

Adam

$$W = W - \alpha \cdot \frac{V_{dW}}{\sqrt{S_{dW} + \varepsilon}}$$

$$B = B - \alpha \cdot \frac{V_{dB}}{\sqrt{S_{dB} + \varepsilon}}$$

Adam proposes these default values:

- The learning rate needs to be tuned.
- For the momentum term β1, a common choice is 0.9.
- For the RMSprop term β2, a common choice is 0.999.
- ε is set to 10–8.

Some key advantages of Adam optimization include the following:

- **Faster convergence**—By adapting the learning rate during training, Adam converges much more quickly than SGD.

- **Easy to implement**—Only requiring first-order gradients, Adam is straightforward to implement and combine with deep neural networks. A few lines of code using Python and PyTorch are all you need.
- **Robust algorithm**—Adam performs well across various model architectures.
- **Little memory requirements**—Adam requires storing just the first and second moments of the gradients, keeping memory needs low.
- **Wide community adoption**—Adam is used extensively by deep learning practitioners and has become a default, go-to optimizer.

Conclusion:

We have looked at different optimization algorithms in neural networks. Considered as a combination of Momentum and RMSProp, Adam is the most superior of them which robustly adapts to large datasets and deep networks. Moreover, it has a straightforward implementation and little memory requirements making it a preferable choice in the majority of situations.

A training iteration, or epoch, is when the model goes a full cycle and sees the entire training dataset at once. The epoch hyperparameter is set to define how many iterations our network continues training. The more training iterations, the more our model learns the features of our training data. To diagnose whether your network needs more or fewer training epochs, keep your eyes on the training and validation error values.

Early stopping:

A training iteration, or epoch, is when the model goes a full cycle and sees the entire training dataset at once. The epoch hyperparameter is set to define how many iterations our network continues training. The more training iterations, the more our model learns the features of our training data.

Early stopping is an algorithm widely used to determine the right time to stop the training process before overfitting happens. It simply monitors the validation error value and stops the training when the value starts to increase.

The good thing about early stopping is that it allows you to worry less about the epochs hyperparameter. You can set a high number of epochs and let the stopping algorithm take care of stopping the training when error stops improving.

1. **Design single unit perceptron for classification of iris dataset without using predefined models.**

   This code implements a basic Perceptron algorithm to classify the first two classes of the Iris dataset (Setosa and Versicolor) using only the first two features. The Perceptron is a type of linear classifier that iteratively adjusts its weights and bias to minimize classification errors.

   Key steps in the code:

   1. **Data Loading and Preprocessing**:
      - The Iris dataset is loaded, but only the first 100 samples and first two features are used. This is because the Perceptron is being trained to distinguish between only two classes (binary classification).
      - The labels are converted from 0 and 1 to -1 and 1 to match the Perceptron's expected output.
   2. **Data Splitting**:
      - The data is split into training and testing sets using an 80-20 split.
   3. **Perceptron Training**:
      - The weights and bias are initialized to zero.
      - The algorithm iterates through the training data for a defined number of epochs, updating the weights and bias whenever a misclassification occurs.
   4. **Testing the Perceptron**:
      - After training, the Perceptron is tested on the test set.
      - The accuracy is calculated based on how many test samples were correctly classified.

   Output:

   The code prints the accuracy of the Perceptron model on the test set. The accuracy percentage indicates how well the model has learned to distinguish between the two classes based on the provided features.

   Potential Improvements:

   - Increase the number of epochs to see if the model improves with more iterations.
   - Implement learning rate decay to potentially improve convergence.
   - Visualize the decision boundary to better understand the model's performance.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

- import numpy as np: This imports the NumPy library, which is useful for numerical operations, especially with arrays.
- from sklearn.datasets import load_iris: This imports the load_iris function from scikit-learn, a machine learning library, to load the Iris dataset.
- from sklearn.model_selection import train_test_split: This imports the train_test_split function from scikit-learn, which is used to split the dataset into training and testing sets.

 # Load the dataset

iris = load_iris()

X = iris.data[:100, :2] # Use only two features and two classes (Setosa and Versicolor)

y = iris.target[:100]

- iris = load_iris(): This loads the Iris dataset and stores it in the variable iris. The dataset contains 150 samples of iris flowers, with four features each and three classes.
- X = iris.data[:100, :2]: This selects the first 100 samples (which belong to the Setosa and Versicolor classes) and only the first two features (sepal length and sepal width) for simplicity.
- y = iris.target[:100]: This selects the corresponding labels (targets) for the first 100 samples.

 # Convert labels to -1 and 1

y = np.where(y == 0, -1, 1)

- y = np.where(y == 0, -1, 1): This converts the class labels from 0 and 1 to -1 and 1 because the Perceptron algorithm typically works with labels of -1 and 1.

 # Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42): This splits the data into training and testing sets. test_size=0.2 means 20% of the data is used for testing, and the rest is used for training. random_state=42 ensures that the split is reproducible.

 # Initialize weights and bias

weights = np.zeros(X_train.shape[1])

bias = 0

learning_rate = 0.1

epochs = 10

- weights = np.zeros(X_train.shape[1]): This initializes the weights to zero. The length of the weights vector matches the number of features (2 in this case).

- bias = 0: This initializes the bias to zero.
- learning_rate = 0.1: This sets the learning rate, which controls how much the weights and bias are updated during training.
- epochs = 10: This sets the number of epochs (complete passes through the training data) for training the Perceptron.

```
# Perceptron training
for epoch in range(epochs):
    for i in range(X_train.shape[0]):
        linear_output = np.dot(X_train[i], weights) + bias
        y_pred = np.where(linear_output > 0, 1, -1)

        # Update weights and bias
        if y_train[i] != y_pred:
            weights += learning_rate * y_train[i] * X_train[i]
            bias += learning_rate * y_train[i]
```

- for epoch in range(epochs): This starts a loop over the number of epochs. Each epoch represents one full pass through the training data.
- for i in range(X_train.shape[0]): This starts a loop over each training sample.
- linear_output = np.dot(X_train[i], weights) + bias: This calculates the linear combination of the input features and the current weights, plus the bias. This is the input to the activation function.
- y_pred = np.where(linear_output > 0, 1, -1): This applies the activation function, which is a step function here. If the linear output is greater than 0, the prediction is 1; otherwise, it is -1.
- if y_train[i] != y_pred: This checks if the predicted label matches the actual label. If not, the weights and bias are updated.
- weights += learning_rate * y_train[i] * X_train[i]: This updates the weights. The adjustment is proportional to the learning rate, the actual label, and the input features.
- bias += learning_rate * y_train[i]: This updates the bias similarly.

```
# Testing the perceptron
correct_predictions = 0
for i in range(X_test.shape[0]):
    linear_output = np.dot(X_test[i], weights) + bias
    y_pred = np.where(linear_output > 0, 1, -1)
```

if y_pred == y_test[i]:

correct_predictions += 1

- correct_predictions = 0: This initializes a counter to keep track of correctly predicted samples.
- for i in range(X_test.shape[0]): This starts a loop over each test sample.
- linear_output = np.dot(X_test[i], weights) + bias: This calculates the linear combination of the input features and the learned weights, plus the bias.
- y_pred = np.where(linear_output > 0, 1, -1): This applies the activation function to get the predicted label.
- if y_pred == y_test[i]: This checks if the predicted label matches the actual label.
- correct_predictions += 1: If the prediction is correct, the counter is incremented.

accuracy = correct_predictions / X_test.shape[0]

print(f"Accuracy: {accuracy * 100:.2f}%")

- accuracy = correct_predictions / X_test.shape[0]: This calculates the accuracy as the ratio of correct predictions to the total number of test samples.
- print(f"Accuracy: {accuracy * 100:.2f}%"): This prints the accuracy as a percentage, formatted to two decimal places.

This code builds and evaluates a simple Perceptron model to classify two types of iris flowers based on two features.

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam, SGD, RMSprop

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean_squared_error
```

- **Importing Libraries:**
    - tensorflow is imported as tf. This is the main library for building and training neural networks.
- Sequential from tensorflow.keras.models is used to build a linear stack of layers.
- Dense from tensorflow.keras.layers is used to create fully connected (dense) layers in the neural network.
- Adam, SGD, RMSprop from tensorflow.keras.optimizers are optimization algorithms (optimizers) used to minimize the loss function.
- EarlyStopping from tensorflow.keras.callbacks is a callback to stop training when the model's performance stops improving on a validation set.
- load_diabetes from sklearn.datasets is used to load the Diabetes dataset, which is a regression problem.
- train_test_split from sklearn.model_selection is used to split the dataset into training and testing sets.
- StandardScaler from sklearn.preprocessing is used to standardize the features (mean = 0, variance = 1).
- mean_squared_error from sklearn.metrics is used to calculate the mean squared error for regression problems.

```
# Load and prepare the data

data = load_diabetes()

X = data.data

y = data.target

# Standardize the features

scaler = StandardScaler()

X = scaler.fit_transform(X)
```

- **Standardizing the Features:**
    - StandardScaler() creates an instance of the StandardScaler.

- fit_transform(X) standardizes the features in X (scales them to have zero mean and unit variance). This step is crucial because neural networks perform better with standardized inputs.

Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

- **Splitting the Data:**
  - train_test_split(X, y, test_size=0.2, random_state=42) splits the dataset into training and testing sets.
  - test_size=0.2 means 20% of the data is reserved for testing, and 80% for training.
  - random_state=42 ensures reproducibility of the split (same data is selected every time the code runs).

Define the model

```
def create_model(activation='relu', optimizer='adam'):
    model = Sequential()
    model.add(Dense(64, input_shape=(X_train.shape[1],), activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(1))  # No activation for regression output
```

Compile the model

```
    model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mse'])
    return model
```

Defining the Model:

- create_model(activation='relu', optimizer='adam') is a function to create an MLP model with customizable activation function and optimizer.
- Sequential() initializes a sequential model, which is a linear stack of layers.
- model.add(Dense(64, input_shape=(X_train.shape[1],), activation=activation)) adds a dense layer with 64 units. The input_shape is set to the number of features in X_train, and the activation function is passed as a parameter.
- model.add(Dense(32, activation=activation)) adds another dense layer with 32 units and the same activation function.
  - model.add(Dense(1)) adds the output layer with a single unit (since this is a regression problem). No activation function is used here because we are predicting a continuous value.

- model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mse']) compiles the model with the specified optimizer and mean squared error as the loss function and evaluation metric.
- The function returns the compiled model.

Define a list of activation functions and optimizers to test

activation_functions = ['relu', 'tanh', 'sigmoid']

optimizers = ['adam', 'sgd', 'rmsprop']

- **Activation Functions and Optimizers:**
  - activation_functions is a list of activation functions to test: 'relu', 'tanh', and 'sigmoid'.
  - optimizers is a list of optimizers to test: 'adam', 'sgd', and 'rmsprop'.

results = {}

- **Results Dictionary:**
  - results is an empty dictionary that will store the Mean Squared

Train and test the model with different configurations

1. **Outer Loop:**

Train and test the model with different configurations

for activation in activation_functions:

- **Outer Loop (Activation Functions):**
  - for activation in activation_functions: starts a loop over the list of activation functions (['relu', 'tanh', 'sigmoid']).
  - This means that for each activation function in the list, the following steps will be executed.

for optimizer in optimizers:

- **Inner Loop (Optimizers):**
  - for optimizer in optimizers: starts another loop, this time over the list of optimizers (['adam', 'sgd', 'rmsprop']).
  - For each activation function selected in the outer loop, this inner loop will iterate over each optimizer, applying the current activation function and optimizer combination to the model.

model = create_model(activation=activation, optimizer=optimizer)

- **Model Creation:**

- model = create_model(activation=activation, optimizer=optimizer) calls the create_model function defined earlier to create a new instance of the MLP model.
- The activation and optimizer parameters passed to create_model correspond to the current iteration's activation function and optimizer.
- This creates a fresh model with the specified activation function and optimizer.

```
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)
```

- **Model Training:**
  - model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0) trains the model on the training data (X_train, y_train).
  - epochs=100 specifies that the model should train for 100 epochs.
  - batch_size=32 specifies that the model should update its weights after every 32 samples.
  - verbose=0 suppresses the output of the training process, so you won't see the progress of each epoch printed out.

Evaluate the model

```
loss, mse = model.evaluate(X_test, y_test, verbose=0)
```

- **Model Evaluation:**
  - loss, mse = model.evaluate(X_test, y_test, verbose=0) evaluates the trained model on the test data (X_test, y_test).
  - evaluate returns the loss (mean squared error in this case) and the metric specified during model compilation (also mean squared error, so the two values are the same here).
  - verbose=0 suppresses the output, so the evaluation process runs silently.

```
results[(activation, optimizer)] = mse
```

- **Store Results:**
  - results[(activation, optimizer)] = mse stores the mean squared error (mse) in the results dictionary.
  - The key in the dictionary is a tuple (activation, optimizer) representing the current activation function and optimizer.
  - The value stored is the mean squared error obtained on the test set for this combination.

```
print(f"Activation: {activation}, Optimizer: {optimizer}, MSE: {mse:.4f}")
```

- **Print Results:**

- print(f"Activation: {activation}, Optimizer: {optimizer}, MSE: {mse:.4f}") prints the current activation function, optimizer, and the corresponding mean squared error (mse) rounded to four decimal places.
- This allows you to see the performance of each combination as the code runs.