CHAPTER 1

Introduction to Computer Vision and Deep Learning

Vision is the best gift to mankind by God.

Right from our birth, vision allows us to develop a conscious mind. Colors, shapes, objects, and faces are all building blocks for our world. This gift of nature is quite central to our senses.

Computer vision is one of the capabilities which allows the machines to replicate this power. And using Deep Learning, we are enhancing our command and making advancements in this field.

This book will examine the concepts of computer vision in the light of Deep Learning. We will study the basic building blocks of Neural Networks, develop pragmatic use cases by taking a case study–based approach, and compare and contrast the performance of various solutions. We will discuss the best practices, share the tips and insights followed in the industry, make you aware of the common pitfalls, and develop a thought process to design Neural Networks.

V. Verdhan, *Computer Vision Using Deep Learning*,
https://doi.org/10.1007/978-1-4842-6616-8_1

Throughout the book, we introduce a concept, explore it in detail, and then develop a use case in Python around it. Since a chapter first builds the foundations of Deep Learning and then its pragmatic usage, the complete knowledge enables you to design a solution and then develop Neural Networks for better decision making.

Some knowledge of Python and object-oriented programming concepts is expected for a good understanding. A basic to intermediate understanding of data science is advisable though not a necessary requirement.

In this introductory chapter, we will develop the concepts of Image Processing using OpenCV and Deep Learning. OpenCV is a great library and is widely used in robotics, face recognition, gesture recognition, AR, and so on. Deep Learning, in addition, offers a higher level of complexity and flexibility to develop Image Processing use cases. We will cover the following topics in this chapter:

(1) Image Processing using OpenCV

(2) Fundamentals of Deep Learning

(3) How Deep Learning works

(4) Popular Deep Learning libraries

1.1 Technical requirements

We are developing all the solutions in Python throughout the book; hence, installation of the latest version of Python is required.

All the code, datasets, and respective results are checked into a code repository at https://github.com/Apress/computer-vision-using-deep-learning/tree/main/Chapter1. You are advised to run all the code with us and replicate the results. This will strengthen your grasp on the concepts.

1.2 Image Processing using OpenCV

An image is also like any other data point. On our computers and mobile phones, it appears as an object or icon in .jpeg, .bmp, and .png formats. Hence, it becomes difficult for humans to visualize it in a row-column structure, like we visualize any other database. Hence, it is often referred to as *unstructured data*.

For our computers and algorithms to analyze an image and work on it, we have to represent an image in the form of integers. Hence, we work on an image pixel by pixel. Mathematically, one of the ways to represent each pixel is the RGB (Red, Green, Blue) value. We use this information to do the Image Processing.

Info The easiest way to get RGB for any color is to open it in Paint in Windows operating system. Hover over any color and get the respective RGB value. In Mac OS, you can use Digital Colour Meter.

Deep Learning allows us to develop use cases which are much more complex to be resolved using traditional Image Processing techniques. For example, detecting a face can be done using OpenCV too, but to be able to recognize one will require Deep Learning.

During the process of developing computer vision solutions using Deep Learning, we prepare our image dataset first. During preparation, we might have to perform grayscaling of the images, detect contours, crop the images, and then feed them to the Neural Network.

OpenCV is the most famous library for such tasks. As a first step, let's develop some building blocks of these Image Processing methods. We will create three solutions using OpenCV.

Note Go to `www.opencv.org` and follow the instructions over there to get OpenCV installed on your system.

The images used for the solutions are the commonly available ones. You are advised to examine the code and follow the step-by-step implementation done. We will detect shape, colors, and a face in an image.

Let's dive into the exciting world of images!

1.2.1 Color detection using OpenCV

When we think of an image, it is made up of shapes, sizes, and colors. Having the capability to detect shape, size, or color in an image can automate a lot of processes and save huge efforts. In the very first example, we are going to develop a "color detection system."

Color detection is having a wide range of utility across domains and industries like manufacturing, automotive, electricity, utilities, and so on. Color detection can be used to look for abruptions, failures, and disruptions to normal behavior. We can train sensors to take a particular decision based on the color and raise an alarm if required.

An image is represented using pixels, and each pixel is made up of RGB values ranging from 0 to 255. We will be using this property to identify the blue color in the image (Figure 1-1). You can change the respective values for the blue color and detect any color of choice.

Follow these steps:

1. Open the Python Jupyter Notebook.
2. First load the necessary libraries, `numpy` and `OpenCV`.

```
import numpy as np
import cv2
```

3. Load the image file.

```
image = cv2.imread('Color.png')
```

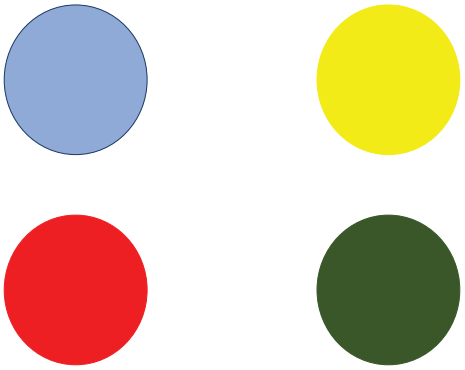

Figure 1-1. *Raw image to be used for color detection. The image shown has four different colors, and the OpenCV solution will detect them individually*

4. Now let us convert our raw image to HSV (Hue Saturation Value) format. It enables us to separate from saturation and pseudo-illumination. `cv2.cvtColor` allows us to do that.

```
hsv_convert = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

5. Define the upper and lower ranges of the color here. We are detecting the blue color. From the `numpy` library, we have given the respective range for the blue color.

```
lower_range = np.array([110,50,50])
upper_range = np.array([130,255,255])
```

6. Now, let's detect the blue color and separate it from the rest of the image.

```
mask_toput = cv2.inRange(hsv_convert, lower_range,
upper_range)
cv2.imshow('image', image)
cv2.imshow('mask', mask_toput)
while(True):
k = cv2.waitKey(5)& 0xFF if k== 27: break
```

The output of this code will be as shown in Figure 1-2.

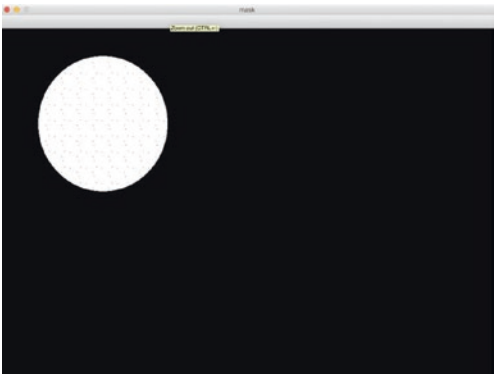

Figure 1-2. *Output of the colour detection system. We want to detect blue colour which is detected and separated from rest of the image*

As visible, the blue color is highlighted in white, while the rest of the image is in black. By changing the ranges in step 5, you can detect different colors of your choice.

With color done, it is time to detect a shape in an image; let's do it!

1.3 Shape detection using OpenCV

Like we detected the blue color in the last section, we will detect triangle, square, rectangle, and circle in an image. Shape detection allows you to separate portions in the image and check for patterns. Color and shape detections make a solution very concrete. The usability lies in safety monitoring, manufacturing lines, automobile centers, and so on.

For Shape detection, we get the contours of each shape, check the number of elements, and then classify accordingly. For example, if this number is three, it is a triangle. In this solution, you will also observe how to grayscale an image and detect contours.

Follow these steps to detect shapes:

1. Import the libraries first.

```
import numpy as np
import cv2
```

2. Load the raw image now shown in Figure 1-3.

```
shape_image = cv2.imread('shape.png')
```

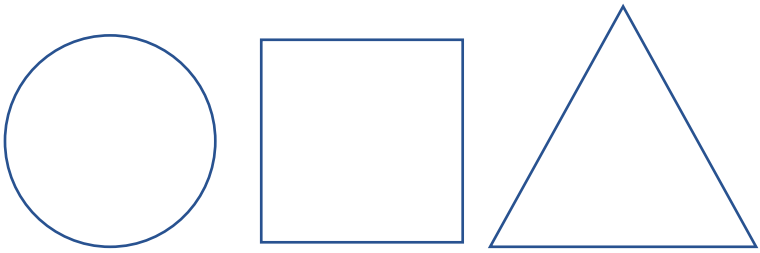

Figure 1-3. *Raw input image for detecting the three shapes of circle, triangle, and rectangle*

3. Convert the image to grayscale next. Grayscaling is done for simplicity since RGB is three-dimensional while grayscale is two-dimensional, and converting to grayscale simplifies the solution. It also makes the code efficient.

```
gray_image = cv2.cvtColor(shape_image, cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(gray_image,127,255,1)
```

4. Find the contours in the image.

```
contours,h = cv2.findContours(thresh,1,2)
```

5. Try to approximate each of the contours using approxPolyDP. This method returns the number of elements in the contours detected. Then we decide the shape based on the number of elements in the contours. If the value is three, it is a triangle; if it's four, it is a square; and so on.

```
for cnt in contours:
    approx =
    cv2.approxPolyDP(cnt,0.01*cv2.arcLength(cnt,True),True)
print (len(approx))
if len(approx)==3:
    print ("triangle")
    cv2.drawContours(shape_image,[cnt],0,(0,255,0),-1)
elif len(approx)==4:
    print ("square")
    cv2.drawContours(shape_image,[cnt],0,(0,0,255),-1)
elif len(approx) > 15:
    print ("circle")
    cv2.drawContours(shape_image,[cnt],0,(0,255,255),-1)
cv2.imshow('shape_image',shape_image)   cv2.waitKey(0)
cv2.destroyAllWindows()
```

6. The output of the preceding code is shown in Figure 1-4.

Figure 1-4. *The output of the color detection system. A circle is shown in yellow, square is shown in red, and triangle is shown in green*

You can now detect shapes in any image. We have detected a circle, a triangle, and a square. A good challenge will be to detect a pentagon or hexagon; are you game?

Let's do something more fun now!

1.3.1 Face detection using OpenCV

Face detection is not a new capability. Whenever we look at a picture, we can recognize a face quite easily. Our mobile phone camera draws square boxes around a face. Or on social media, a square box is created around a face. It is called *face detection*.

Face detection refers to locating human faces in digital images. Face detection is different from face recognition. In the former, we are only detecting a face in an image, whereas in the latter we are putting a name to the face too, that is, who is the person in the photo.

Most of the modern cameras and mobiles have a built-in capability to detect faces. A similar solution can be developed using OpenCV. It is simpler to understand and implement and is built using the `Haar-cascade` algorithm. We will highlight faces and eyes in a photograph while using this algorithm in Python.

`Haar-cascade classifier` is used to detect faces and other facial attributes like eyes in an image. It is a Machine Learning solution wherein training is done on a lot of images which have a face and which do not have a face in them. The classifier learns the respective features. Then we use the same classifier to detect faces for us. We need not do any training here as the classifier is already trained and ready to be used. Saves time and effort, too!

Info Object detection using Haar-based cascade classifiers was proposed by Paul Viola and Michael Jones in their paper "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. You are advised to go through this path-breaking paper.

Follow these steps to detect a face:

1. Import the libraries first.

```
import numpy as np
import cv2
```

2. Load the classifier xml file. The .xml file is designed by OpenCV and is created by training cascade of negative faces superimposed on positive images and hence can detect the facial features.

```
face_cascade = cv2.CascadeClassifier(
'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(
'haarcascade_eye.xml')
```

3. Next, load the image (Figure 1-5).

```
img = cv2.imread('Vaibhav.jpg')
```

Figure 1-5. *Raw input image of a face which is used for the face detection using the Haar-cascade solution*

4. Convert the image to grayscale.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

5. Execute the following code to detect a face in the image. If any face is found, we return the position of the detected face as Rect(x,y,w,h). Subsequently, the eyes are detected on the face.

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
image = cv2.rectangle(image,(x,y),(x+w,y+h),
(255,0,0),2) roi_gr = gray[y:y+h, x:x+w]
roi_clr = img[y:y+h, x:x+w]
the_eyes = eye_cascade.detectMultiScale(roi_gr)
for (ex,ey,ew,eh) in the_eyes:
    cv2.rectangle(roi_clr,(ex,ey),(ex+ew,ey+eh),
    (0,255,0),2)
    cv2.imshow('img',image) cv2.waitKey(0)
    cv2.destroyAllWindows()
```

6. The output is shown in Figure 1-6. Have a look how a blue box is drawn around the face and two green small boxes are around the eyes.

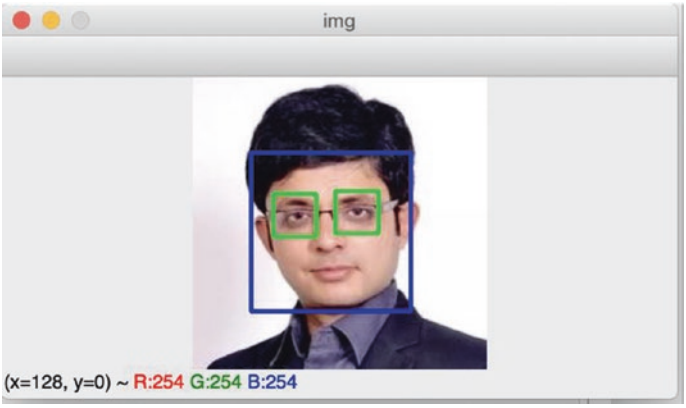

Figure 1-6. *Face and eyes detected in the image; a green box is around the eyes and blue square around the face*

Face detection allows us to find faces in images and videos. It is the first step for face recognition. It is used widely for security applications, attendance monitoring, and so on. We will be developing face detection and recognition using Deep Learning in subsequent chapters.

We have studied some of the concepts of Image Processing. It is time to examine and learn the concepts of Deep Learning. These are the building blocks for the journey you have embarked upon.

1.4 Fundamentals of Deep Learning

Deep Learning is a subfield of Machine Learning. The "deep" in Deep Learning is having successive layers of representation; hence, the depth of the model refers to the number of layers in the Artificial Neural Network (ANN) model and is essentially referred to as Deep Learning.

It is a novel approach to analyze historical data and learn from successive layers of increasingly meaningful representations. The typical process in a Deep Learning project is similar to a Machine Learning project as described in the following and shown in Figure 1-7.

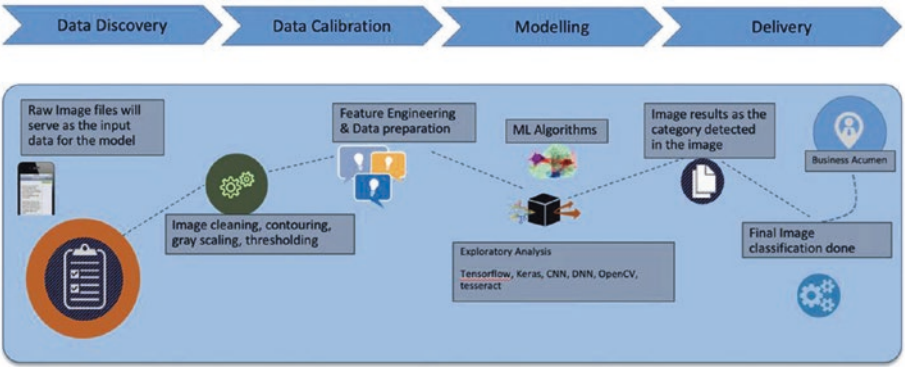

Figure 1-7. *End-to-end machine learning process from data discovery to the final development of the solution. All the steps are discussed in detail here and are again revisited in Chapter 8 of the book*

1. Data ingestion: Raw data files/image/text and so on are ingested into the system. They serve as the input data to train and test the network.
2. Data cleaning: In this step, we clean the data. Often, there is too much noise like junk values, duplicates, NULL, and outliers present in a structured dataset. All such data points have to be treated at this stage. For images, we might have to remove the unnecessary noise in the images.
3. Data preparation: We make our data ready for training. In this step, new derived variables might be required, or we might need to rotate/crop images in case we are working on image datasets.
4. Exploratory data analysis: We perform initial analysis to generate quick insights about our datasets.
5. Network design and training the model: We design our Neural Network here and decide on the number of hidden layers, nodes, activation functions, loss functions, and so on. The network is then trained.
6. Check the accuracy and iterate: We measure the accuracy of the network. We use the Confusion Matrix, AUC value, precision, recall, and so on to measure. Then we tune the hyperparameters and tune further.
7. The final model is presented to the business and we get the feedback.
8. We iterate the model and improve it based on the feedback received and a final solution is created.
9. The model is deployed to production. It is then maintained and refreshed at regular intervals.

These steps are typically followed during a Machine Learning project. We will be examining all these steps in great details in the last chapter of this book. It is now a good time to understand Neural Networks.

1.4.1 The motivation behind Neural Network

Artificial Neural Networks (ANNs) are said to be inspired by the way a human brain works. When we see a picture, we associate a label against it. We train our brain and senses to recognize a picture when we see it again and label it correctly.

ANN learns to perform similar tasks by learning or getting trained. This is done by looking at various examples of historical data points like transactional data or images and most of the time without being programmed for specific rules. For example, to distinguish between a car and a man, an ANN will start with no prior understanding and knowledge of the attributes of each of the class. It then generates attributes and the identification characteristics from the training data. It then learns those attributes and uses them later to make predictions.

Formally put, "learning" in the context of Artificial Neural Networks refers to adjusting the weights and the bias inside the network to improve the subsequent accuracy for the network. And an obvious way to do this is to reduce the error term which is nothing but the difference between the actual value and predicted value. To measure the error rate, we have a cost function defined which is rigorously evaluated during the learning phase of the network. We will be examining all these terms in detail in the next section.

A typical Neural Network looks like Figure 1-8.

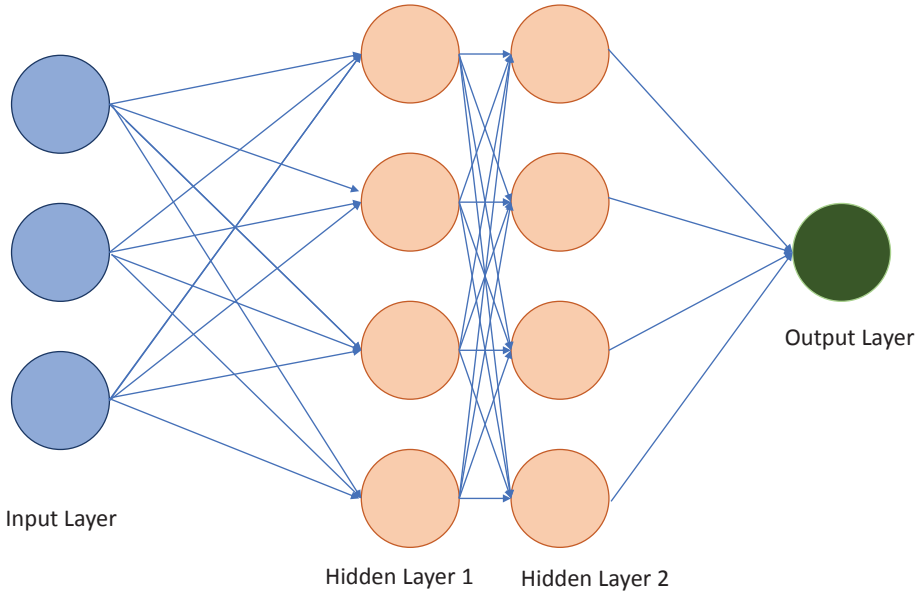

Figure 1-8. *A typical Neural Network with an input layer, hidden layers, and an output layer. Each layer has a few neurons inside it. Hidden layers act as the heart and soul of the network. The input layer accepts the input data, and the output layer is responsible for generating the final results*

The Neural Network shown earlier has three input units, two hidden layers with four neurons each, and one final output layer.

Now let us discuss various components of a Neural Network in subsequent sections.

1.4.2 Layers in a Neural Network

A basic Neural Network architecture consists of predominantly three layers:

- Input layer: As the name signifies, it receives the input data. Consider feeding raw images/processed images to the input layer. This is the first step of a Neural Network.

- Hidden layers: They are the heart and soul of the network. All the processing, feature extraction, learning, and training are done in these layers. Hidden layers break the raw data into attributes and features and learn the nuances of the data. This learning is used later in the output layer to make a decision.
- Output layer: The decision layer and final piece in a network. It accepts the outputs from the preceding hidden layers and then makes a judgment on the final classification.

The most granular building block of a network is a neuron. A neuron is where the entire magic takes place, which we are discussing next.

1.4.3 Neuron

A neuron or artificial neurons are the foundation of a Neural Network. The entire complex calculation takes place in a neuron only. A layer in the network can contain more than one neuron.

A neuron receives input from the previous layers or the input layers and then does the processing of the information and shares an output. The input data can be the raw data or processed information from a preceding neuron. The neuron then combines the input with their own internal state and reaches a value using an activation function (we'll discuss activation functions in a while). Subsequently, an output is generated using the output function.

A neuron can be thought of as Figure 1-9 where it receives the respective inputs and calculates the output.

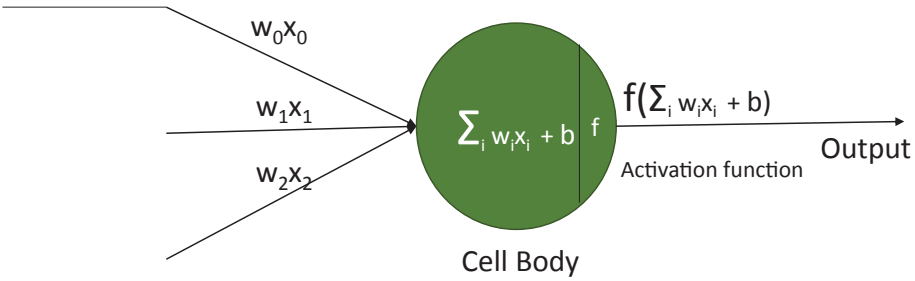

Figure 1-9. *A representation of a neuron receiving input from the previous layers and using the activation function to process and give the output. It is the building block of a Neural Network*

The input to a neuron is received from the output of its predecessors and their respective connections. The input received is calculated as a weighted sum, and a bias term is generally added too. This is the function of a *propagation function*. As shown in Figure 1-9, f is the activation function, w is the weight terms, and b is the bias term. After the calculations are done, we receive the output.

For example, the input training data will have raw images or processed images. These images will be fed to the input layer. The data now travels to the hidden layers where all the calculations are done. These calculations are done by neurons in each layer.

The output is the task that needs to be accomplished, for example, identification of an object or if we want to classify an image and so on.

Like we discussed, to a large extent, Neural Networks are able to extract the information themselves, but still we have to initiate a few parameters for the process of training the network. They are referred to as *hyperparameters* which we are discussing next.

1.4.4 Hyperparameters

During training a network, the algorithm is constantly learning the attributes of the raw data. But there are a few parameters which a network

cannot learn itself and requires initial settings. Hyperparameters are those variables and attributes which an Artificial Neural Network cannot learn by itself. These are the variables that determine the structure of the Neural Network and the respective variables which are useful to train the network.

Hyperparameters are set before the actual training of the network. The learning rate, number of hidden layers in the network, number of neurons in each layer, activation function, number of epoch, batch size, dropout, and network weight initialization are examples of hyperparameters.

Tuning the hyperparameters is the process of choosing the best value for the hyperparameters based on its performance. We measure the performance of the network on the validation set and then tweak the hyperparameters and then reevaluate and retweak, and this process continues. We will examine various hyperparameters in the next section.

1.4.5 Connections and weight of ANN

The ANN consists of various connections. Each of the connections aims to receive the input and provide the computed output. This output serves as an input to the next neuron.

Also, each connection is assigned a weight which is a representative of its respective importance. It is important to note that a neuron can have multiple input and output connections which means it can receive inputs and deliver multiple signals.

The next term is again an important component – the bias term.

1.4.6 Bias term

Bias is just like adding an intercept value to a linear equation. It is an extra or additional parameter in a network.

The simplest way to understand bias is as per the following equation:

$$y = mx + c$$

If we do not have the constant term c, the equation will pass through (0,0). If we have the constant term c, we can expect a better fitting machine learning model.

As we can observe in Figure 1-10, on the left, we have a neuron without a bias term, while on the right we have added a bias term. So the bias allows us to adjust the output along with the weighted sum of the inputs.

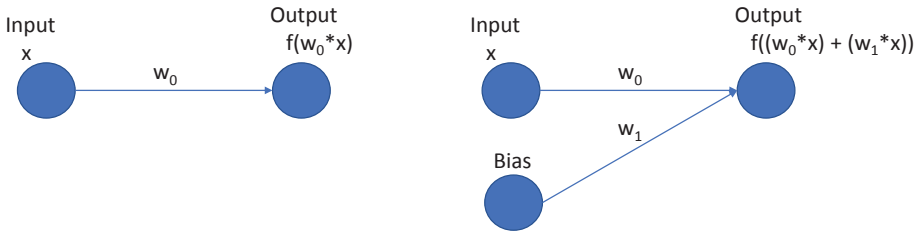

Figure 1-10. *A bias term helps in better fitting the model. On the left side, there is no bias term, and on the right we have the bias term. Note that the bias term has a weight associated with it*

The bias term hence acts like a constant term in a linear equation, and it helps in fitting the data better.

We will now study one of the most important attributes in a Neural Network – activation functions – in the next section.

1.4.7 Activation functions

The primary role of an activation function is to decide whether a neuron/perceptron should fire or not. They play a central role in adjusting the gradients during the training of the network at a later stage. An activation function is shown in Figure 1-9. They are sometimes referred to as *transfer functions*.

The nonlinear behavior of activation functions allows Deep Learning networks to learn complex behaviors. You will examine what is meant by nonlinear behaviors in Chapter 2. We will study some of the commonly used functions now.

1.4.7.1 Sigmoid function

The Sigmoid function is a bounded monotonic mathematical function. It is a differentiable function with an S-shaped curve, and its first derivative function is bell shaped. It has a nonnegative derivative function and is defined for all real input values. The Sigmoid function is used if the output value of a neuron is between 0 and 1.

Mathematically, a sigmoid function is as shown in Equation 1-1.

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^{x}}{e^{x}+1} \qquad \text{(Equation 1-1)}$$

The graph of a sigmoid function can be seen in Figure 1-11. Note the shape of the function and the max and min values.

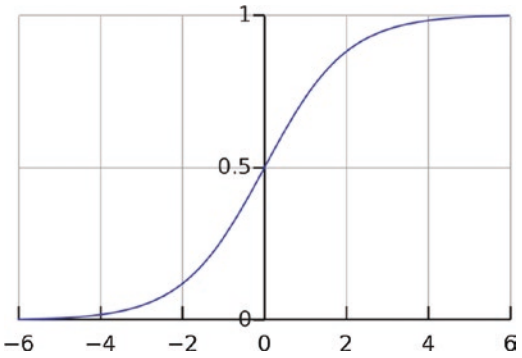

Figure 1-11. *A Sigmoid function; note it is not zero centered, and its values lie between 0 and 1. A sigmoid suffers from the problem of vanishing gradients*

A Sigmoid function finds its applications in complex learning systems. Often you would find a Sigmoid function is being used when a specific mathematical model is not fitting. It is usually used for binary classification and in the final output layer of the network. A Sigmoid function suffers from a vanishing gradient problem which we will discuss in subsequent sections.

1.4.7.2 tanh function

In mathematics, the tangent hyperbolic function or tanh is a differentiable hyperbolic function. It is a scaled version of the Sigmoid function. It is a smooth function, and its input values are in the range of –1 to +1.

With more stable gradients, it has fewer vanishing gradient problems than the Sigmoid function. A tanh function can be represented as Figure 1-12 and can be seen in Equation 1-2.

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad \text{(Equation 1-2)}$$

A graphical representation of tanh is also shown. Note the difference between the Sigmoid and tanh function. Observe how tanh is a scaled version of the Sigmoid function.

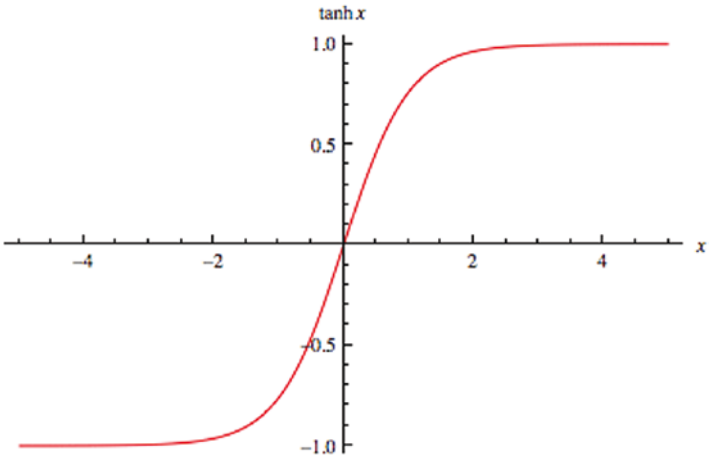

Figure 1-12. *A tanh function; note it passes through zero and is a scaled version of the sigmoid function. Its value lies between –1 and +1. Similar to sigmoid, tanh suffers from a vanishing gradient problem too*

A tanh function is generally used in the hidden layers. It makes the mean closer to zero which makes the training easier for the next layer in the network. This is also referred to as *centering the data*. A tanh function can be derived from the Sigmoid function and vice versa. Similar to sigmoid, the tanh function suffers from a vanishing gradient problem which we will discuss in subsequent sections.

We now examine the most popular activation function – ReLU.

1.4.7.3 Rectified Linear Unit or ReLU

The Rectified Linear Unit or ReLU is an activation function that defines the positives of an argument.

ReLU is a simple function, least expensive to compute, and can be trained much faster. It is unbounded and not centered at zero. It is differentiable at all the places except zero. Since a ReLU function can be trained much faster, you will find it to be used more frequently.

We can examine the ReLU function and the graph in Figure 1-13. The equation can be seen in Equation 1-3. Note that the value is 0 even for the negative values, and from 0 the value starts to incline.

$F(x) = \max(0, x)$ i.e. will give output as x if positive else 0 (Equation 1-3)

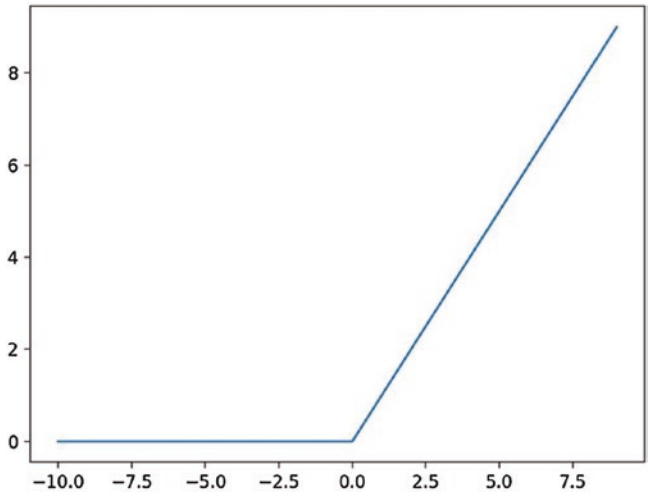

Figure 1-13. *A ReLU function; note it is quite simple to compute and hence faster in training. It is used in the hidden layers of the network. A ReLU is faster to train than sigmoid and tanh*

Since the ReLU function is less complex, is computationally less expensive, and hence is widely used in the hidden layers to train the networks faster, we will also use ReLU while designing the network. Now we study the softmax function which is used in the final layer of a network.

1.4.7.4 Softmax function

The softmax function is used in the final layer of the Neural Network to generate the output from the network. The output can be a final classification of an image for distinct categories.

The softmax function calculates the probabilities for each of the target classes over all the possibilities. It is an activation function that is useful for multiclass classification problems and forces the Neural Network to output the sum of 1.

As an example, if the input is [1,2,3,4,4,3,2,1] and we take a softmax, then the corresponding output will be [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175]. This output is allocating the highest weight to the highest

value which is 4 in this case. And hence it can be used to highlight the highest value. A more practical example will be if the number of distinct classes for an image are cars, bikes, or trucks, the softmax function will generate three probabilities for each category. The category which has received the highest probability will be the predicted category.

We have studied the important activation functions. But there can be other activation functions like Leaky ReLU, ELU, and so on. We will encounter these activation functions throughout the book. Table 1-1 shows a summary of the activation functions for quick reference.

Table 1-1. *The major activation functions and their respective details*

Activation Function	Value	Positives	Challenges
Sigmoid	[0,1]	(1) Nonlinear (2) Easy to work with (3) Continuous differentiable (4) Monotonic and does not blow up the activations	(1) Output not zero centered (2) Problem of vanishing gradients (3) Is slow to train
tanh	[-1,1]	(1) Similar to the sigmoid function (2) Gradient is stronger but is preferred over sigmoid	(1) Problem of vanishing gradient

(*continued*)

Table 1-1. (*continued*)

Activation Function	Value	Positives	Challenges
ReLU	[0,inf]	(1) Not linear (2) Easy to compute and hence fast to train (3) Resolves problem of vanishing gradient	(1) Used only in the hidden layers (2) Can blow up the activations (3) For the x<0 region, the gradient will be zero. Hence, weights do not get updated (dying ReLU problem)
Leaky ReLU	max(0,x)	(1) A variant of ReLU (2) Fixes dying ReLU problem	(1) Cannot be used for complex classifications
ELU	[0,inf]	(1) Alternative to ReLU (2) The output is smoother	(1) Can blow up the activations
Softmax	Calculates probabilities	Generally used in the output layer	

Activation functions form the core building block of a network. In the next section, we discuss the learning rate which guides how the network is going to learn and optimize the training.

1.4.8 Learning rate

For a Neural Network, the *learning rate* will define the size of the corrective steps which a model takes to reduce the errors. A higher learning rate has lower accuracy but a shorter time to train, while a lower learning rate will take a long time to train but have higher accuracy. You have to arrive at the most optimized value of it.

Specifically put, learning rate will govern the adjustments to be made to the weights during training of the network. Learning rate will directly impact the amount of time it will take for the network to converge and reach the global minima. In most of the cases, having a learning rate of 0.01 is acceptable.

We will now explore perhaps the most important process in the training process – the backpropagation algorithm.

1.4.9 Backpropagation

We studied about the learning rate in the last section. The objective of the training process is to reduce the error in the predictions. While the learning rate defines the size of the corrective steps to reduce the error, backpropagation is used to adjust the connection weights. These weights are updated backward based on the error. Following it, the errors are recalculated, gradient descent is calculated, and the respective weights are adjusted.

Figure 1-14 shows the process for backpropagation where the information flows from the output layer back to the hidden layers. Note that the flow of information is backwards as compared to forward propagation where the information flows from left to right.

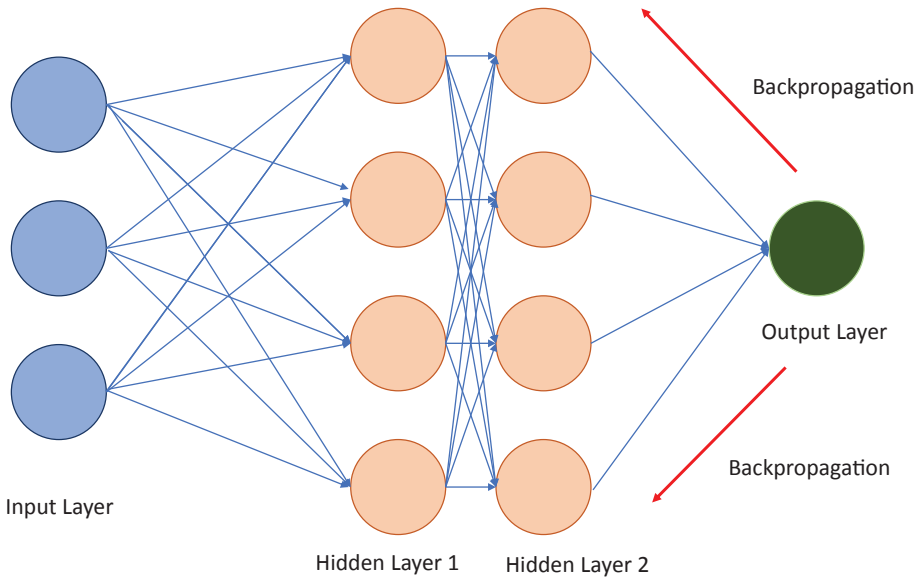

Figure 1-14. *Backpropagation in a Neural Network. Based on the error the information flows from the output backward, and subsequently weights are recalculated*

Let us explore the process in more detail.

Once the network makes a prediction, we can calculate the error which is the difference between the expected and the predicted value. This is referred to as the cost function. Based on the value of cost, the Neural Network then adjusts its weights and biases to get closest to the actual values or, in other words, minimize the error. This is done during backpropagation.

Note You are advised to refresh differential calculus to understand the backpropagation algorithm better.

During backpropagation, the parameters of the connections are repeatedly and iteratively updated and adjusted. The level of adjustments is determined by the gradient of the cost function with respect to those parameters. The gradient will tell us in which direction should we adjust the

weights to minimize the cost. And these gradients are calculated using the chain rule. Using the chain rule, the gradients are calculated for one layer at a time, iterating backward from the last layer to the first layer. This is done to avoid redundant calculations of intermediate terms in the chain rule.

Sometimes, we encounter the problem of vanishing gradients during the training of a Neural Network. The vanishing gradient problem is the phenomenon when the initial layers of the network cease to learn as the gradient becomes close to zero. It makes the network unstable, and the initial layers of the network will not be able to learn anything. We will again explore vanishing gradients in Chapters 6 and 8.

We now discuss the issue of overfitting – one of the most common problems faced during training.

1.4.10 Overfitting

You know that training data is used by a network to learn the attributes and patterns. But we want our Machine Learning models to perform well on unseen data so that we can use it to make the predictions.

To measure the accuracy of Machine Learning, we have to evaluate the performance of both training and testing datasets. Often, the network mimics the training data well and gets good training accuracy, whereas on the testing/validation dataset, the accuracy drops. This is called *overfitting*. Simply put, if the network is working well on the training dataset but not so great on unseen dataset, it is called overfitting.

Overfitting is a nuisance, and we have to fight it, right? To tackle overfitting, you can train your network with more training data. Or reduce the networks' complexity. By reducing the complexity, we suggest to reduce the number of weights, the value of weights, or the structure of the network itself.

Batch normalization and *Dropout* are two other techniques to mitigate the problem of overfitting.

Batch normalization is a type of regularization method. It is the process of normalizing the output from a layer with zero mean and

a standard deviation of one. It reduces the emphasis on the weight initialization and hence reduces overfitting.

Dropout is another technique to fight the problem of overfitting. It is a regularization method. During training, the output of some layers is randomly dropped out or neglected. The effect is that we get different neural nets for each combination. It makes the training process noisy too. Figure 1-15 represents the impact of Dropout. The first figure on the left (Figure 1-15(i)) is a standard Neural Network. The one on the right (Figure 1-15(ii)) is the result after Dropout.

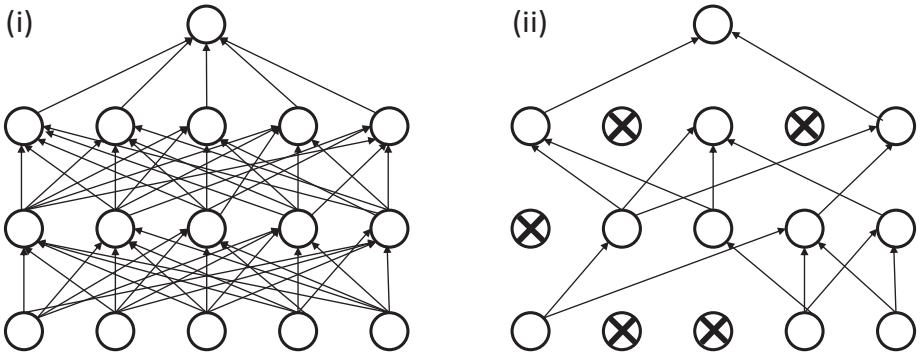

Figure 1-15. *Before dropout, the network might suffer from overfitting. After dropout, random connections and neurons are removed, and hence the network will not suffer from overfitting*

With dropout, we can tackle overfitting in our network and reach a much more robust solution.

Next, we will study the process of optimization using gradient descent.

1.4.11 Gradient descent

The purpose of a Machine Learning solution is to find the most optimum value for our. We want to decrease the loss during the training phase or maximize the accuracy. Gradient descent can help to achieve this purpose.

Gradient descent is used to find the global minimum or global maximum of a function. It is a highly used optimization technique. We proceed in the direction of the steepest descent iteratively which is defined by the negative of the gradient.

But gradient descent can be slow to run on very large datasets. It is due to the fact that one iteration of the gradient descent algorithm predicts for every instance in the training dataset. Hence, it is obvious that it will take a lot of time if we have thousands of records. For such a situation, we have *Stochastic Gradient Descent*.

In Stochastic Gradient Descent, rather than at the end of the batch, the coefficients are updated for each training instance, and hence it takes less time.

Figure 1-16 shows the way a gradient descent works. Notice how we can progress downward toward the global minimum.

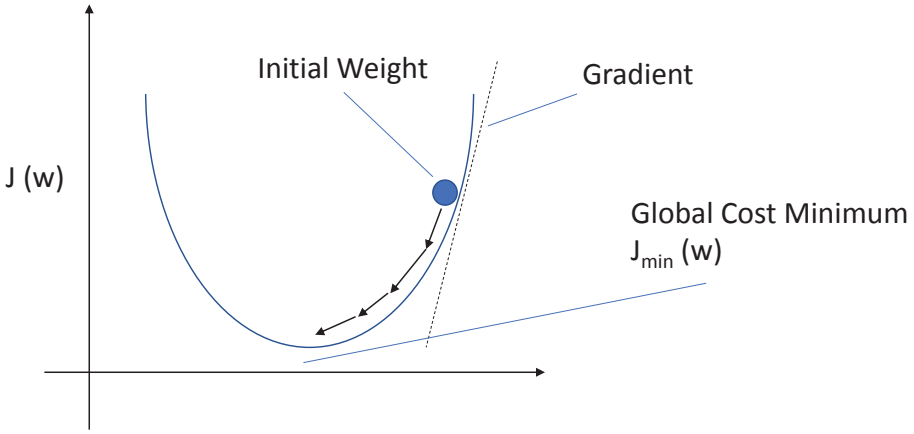

Figure 1-16. *The concept of gradient descent; notice how it aims to reach the global minimum. The objective of training the network is to minimize the error encountered*

We examined how we optimize the function using gradient descent. The way to check for the efficacy of the machine learning model is to measure how far or how close the predictions are to the actual values. And that is defined using loss which we discuss now.

1.4.12 Loss functions

Loss is the measure of our model's accuracy. In simple terms, it is the difference of actual and predicted values. The function which is used to calculate this loss is called the *loss function*.

Different loss functions give different values for the same loss. And since the loss is different, the respective model's performance will differ too.

We have different loss functions for regression and classification problems. Cross-entropy is used to define a loss function for the optimization. To measure the error between the actual output and the desired output, generally, the mean squared error is used. Some researchers suggest to use the cross-entropy error instead of the mean squared error. Table 1-2 gives a quick summary for different loss functions.

Table 1-2. *The various loss functions which can be used with their respective equations and the usage*

Loss Function	Equation for the Loss	Used for
Cross-entropy	-y(log(p) + (1-y) log(1-p))	Classification
Hinge loss	max(0, 1- y *f(x))	Classification
Absolute error	|y - f(x)|	Regression
Squared error	$(y - f(x))^2$	Regression
Huber loss	$L_\delta = ½ (y - f(x))^2$, if |y-f(x) <= δ else δ|y-f(x)| - ½ δ^2	Regression

We have now examined the main concepts of Deep Learning. Now let us study how a Neural Network works. We will understand how the various layers interact with each other and how information is passed from one layer to another.

Let's get started!

1.5 How Deep Learning works?

You now know that a Deep Learning network has various layers. And you have also gone through the concepts of Deep Learning. You may be wondering how these pieces come together and orchestrate the entire learning. The entire process can be examined as follows:

Step 1

You may be wondering what a layer actually does, what a layer achieves, and what is stored in the respective weights of a layer. It is nothing but a set of numbers. Technically, it is imperative that the transformation implemented by a layer is parameterized by its weights which are also referred to as parameters of a layer.

And what is meant by learning is the next question. Learning for a Neural Network is finding the best combination and values for weights for all the layers of the network so that we can achieve the best accuracy. As deep Neural Networks can have many values of such parameters, we have to find the most optimum value for all the parameters. This seems like a herculean task considering that changing one value impacts others.

Let us show the process in the Neural Network by means of a diagram (Figure 1-17). Examine our input data layer. Two data transformation layers each having respective weights associated with them. And then we have a final prediction for the target variable Y.

The images are fed into the input layer and then are transformed in the data transformation layers.

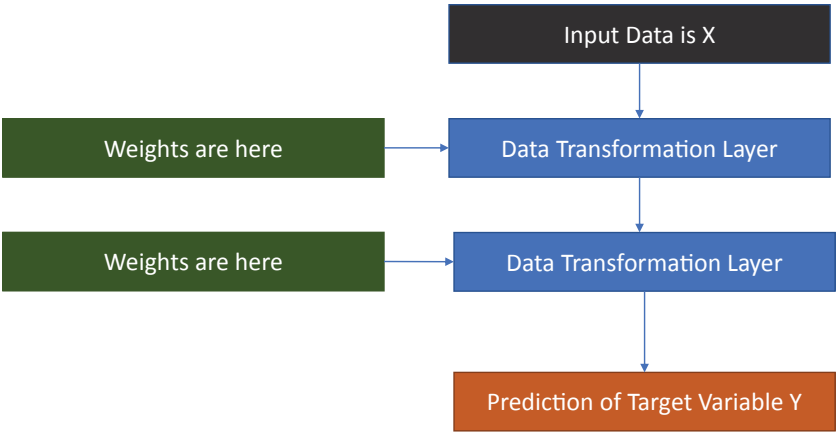

Figure 1-17. *The input data is transformed in the data transformation layers, weights are defined, and the initial prediction is made*

Step 2

We have created a basic skeleton in step 1. Now we have to gauge the accuracy of this network.

We would want to control the output of a Neural Network; we have to compare and contrast the accuracy of the output.

Info Accuracy will refer to how far our prediction is from the actual value. Simply put, how good or bad our predictions are from the real values is a measure of accuracy.

Accuracy measurement is done by the loss function of the network, also called the *objective function*. The loss function takes the predictions of the network and the true or actual target values. These actual values are what we expected the network to output. The loss function computes a distance score, capturing how well the network has done.

Let's update the diagram we created in step 1 by adding a loss function and corresponding loss score (Figure 1-18). It helps to measure the accuracy for the network.

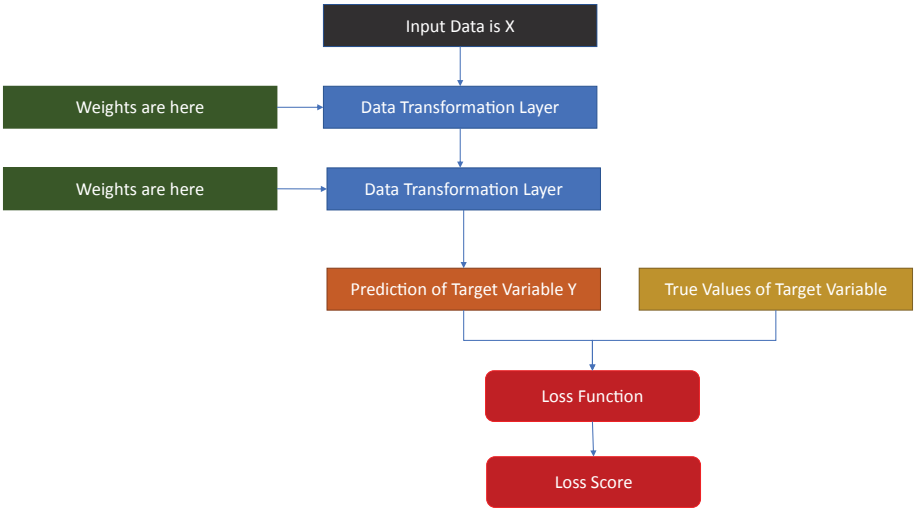

Figure 1-18. *Add a loss function to measure the accuracy; loss is the difference between actual and predicted values. At this stage, we know the performance of the network based on the error term*

Step 3

As we discussed earlier, we have to maximize the accuracy or lower the loss. This will make the solution robust and accurate in predictions.

In order to constantly lower the loss, the score (predictions – actual) is then used as a feedback signal to adjust the value of the weights a little which is done by the optimizer. This task is done by the *backpropagation algorithm* which is sometimes called the central algorithm in Deep Learning.

Initially, some random values are assigned to the weights, so the network is implementing a series of random transformations. And logically enough, the output is quite different from what we would be expecting, and the loss score is accordingly very high. After all, it is the very first attempt!

But this is going to improve. While training the Neural Network, we constantly encounter new training examples. And with each fresh example, the weights are adjusted a little in the correct direction, and subsequently the loss score decreases. We iterate this training loop many times, and it results in most optimum weight values that minimize the loss function.

We then achieve our objective which is a trained network with minimal loss. It means that actual and predicted values are very close to each other. To achieve the complete solution, we scale up this mechanism which is visible in Figure 1-19.

Notice the optimizer which provides regular and continuous feedback to reach the best solution.

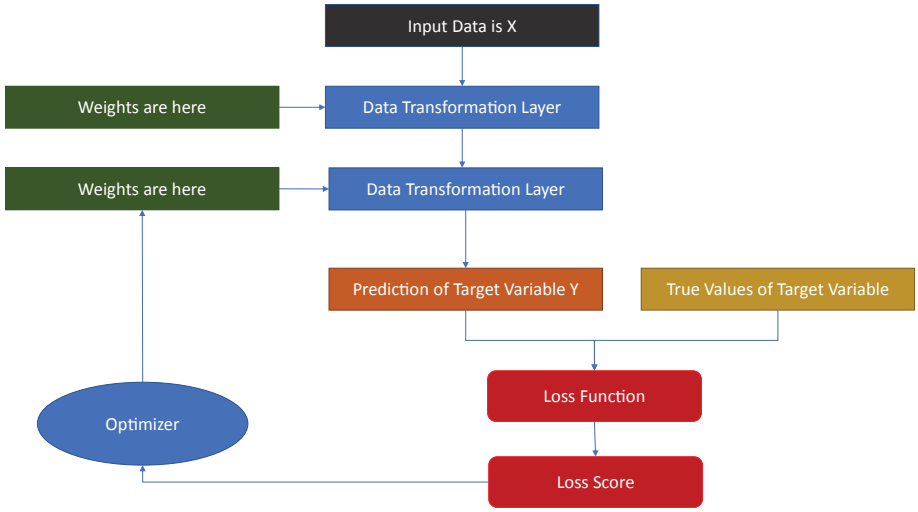

Figure 1-19. *Optimizer to give the feedback and optimize the weights; this is the process of backpropagation. It ensures that the error is reduced iteratively*

Once we have achieved the best values for our network, we call that our network is trained now. We can now use it to make predictions on unseen datasets.

Now you have understood what the various components of Deep Learning are and how they work together. It is time for you to now examine all the tools and libraries for Deep Learning.

1.5.1 Popular Deep Learning libraries

There are quite a few Deep Learning libraries which are available. These packages allow us to develop solutions faster and with minimum efforts as most of the heavy lifting is done by these libraries.

We are discussing the most popular libraries here.

TensorFlow: TensorFlow (TF) developed by Google is arguably one of the most popular and widely used Deep Learning frameworks. It was launched in 2015 and since is being used by a number of businesses and brands across the globe.

Python is mostly used for TF, but C++, Java, C#, JavaScript, and Julia can also be used. You have to install the TF library on your system and import the library. And it is ready to be used!

Note Go to `www.tensorflow.org/install` and follow the instructions to install TensorFlow.

A TF model has to be retrained in case of any modifications to the model architecture. It operates with a static computation graph which means we define the graph first, and then the calculations are run.

It is quite popular as it's developed by Google. It can work on mobile devices like iOS and Android too.

Keras: It is one of the easiest Deep Learning frameworks for starters and fantastic for understanding and prototyping simple concepts. Keras was initially released in 2015 and is one of the most recommended libraries to understand the nuances of Neural Networks.

Note Go to `https://keras.io` and follow the instructions to install Keras. Tf.keras can be used as an API and will be used frequently in this book.

It is a mature API-driven solution. Prototyping in Keras is facilitated to the limit. Serialization/deserialization APIs, callbacks, and data streaming using Python generators are very mature. Massive models in Keras are reduced to single-line functions which makes it a less configurable environment.

PyTorch: Facebook's brain-child PyTorch was released in 2016 and is one of the popular Deep Learning libraries. We can use debuggers in PyTorch, for example, pdb or PyCharm. PyTorch operates with dynamically updated graphs and allows data parallelism and distributed learning models. For small projects and prototyping, PyTorch should be your choice; however, for cross-platform solutions, TensorFlow is known to be better.

Sonnet: DeepMind's Sonnet is developed using and on top of TF. Sonnet is designed for complex Neural Network applications and architectures.

Sonnet creates primary Python objects which correspond to a particular part of the Neural Network (NN). After this, these Python objects are connected independently to the computational TensorFlow graph. It simplifies the design which is due to the separation of the process of creating objects and associating them with a graph. Further, the capability to have high-level object-oriented libraries is advantageous as it helps in abstraction when we develop Machine Learning algorithms.

MXNet: Apache's MXNet is a highly scalable Deep Learning tool that is easy to use and has detailed documentation. A large number of languages like C++, Python, R, Julia, JavaScript, Scala, Go, and Perl are supported by MXNet.

There are other frameworks too like Swift, Gluon, Chainer, DL4J, and so on; however, we've only discussed the popular ones in this book. Table 1-3 gives an overview of all the frameworks.

Table 1-3. *Major Deep Learning frameworks and their respective attributes*

Framework	Source	Attributes
TensorFlow	Open source	Most popular, can work on mobile too, TensorBoard provides visualizations
Keras	Open source	API-driven mature solution, very easy to use
PyTorch	Open source	Allows data parallelism and very good for quick product building
Sonnet	Open source	Simplified design, creates high-level object
MXNet	Open source	Highly scalable, easy to use
MATLAB	Licensed	Highly configurable, provides deployment capabilities

1.6 Summary

Deep Learning is a continuous learning experience and requires discipline, rigor, and commitment. You have taken the first step in your learning journey. In this first chapter, we studied concepts of Image Processing and Deep Learning. They are the building blocks for the entire book and your path ahead. We developed three solutions using OpenCV.

In the next chapter, you will deep dive into TensorFlow and Keras. And you will develop your first solution using Convolutional Neural Network. Right from designing the network, training it, and implementing it. So stay focused!

REVIEW EXERCISES

1. What are the various steps in Image Processing?
2. Develop an object detection solution using OpenCV.
3. What is the process of training a Deep Learning network?
4. What is overfitting and how do we tackle it?
5. What are various activation functions?

1.6.1 Further readings

1. A brief introduction to OpenCV: https://ieeexplore.ieee.org/document/6240859.
2. OpenCV for computer vision applications: www.researchgate.net/publication/301590571_OpenCV_for_Computer_Vision_Applications.
3. OpenCV documentation can be accessed at https://docs.opencv.org/.
4. Go through these following papers:
   a. www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
   b. https://arxiv.org/pdf/1404.7828.pdf

CHAPTER 2

Nuts and Bolts of Deep Learning for Computer Vision

The mind is not a vessel to be filled, but a fire to be kindled.

—Plutarch

We humans are blessed with extraordinary powers of mind. These powers allow us to differentiate and distinguish, develop new skills, learn new arts, and make rational decisions. Our visual powers have no limits. We can recognize faces regardless of pose and background. We can distinguish objects like cars, dogs, tables, phones, and so on irrespective of the brand and type. We can recognize colors and shapes and distinguish clearly and easily between them. This power is developed periodically and systematically. In our young age, we continuously learn the attributes of objects and develop our knowledge. That information is kept safe in our memory. With time, this knowledge and learning improve. This is such an astonishing process that iteratively trains our eyes and minds. It is often argued that Deep Learning originated as a mechanism to mimic these extraordinary powers. In computer vision, Deep Learning is helping us to uncover the capabilities which can be used to help organizations use

V. Verdhan, *Computer Vision Using Deep Learning*,
https://doi.org/10.1007/978-1-4842-6616-8_2

computer vision for productive purposes. Deep Learning has evolved a lot and is still having a lot of scope for further progress.

In the first chapter, we started with fundamentals of Deep Learning. In this second chapter, we will build on those fundamentals, go deeper, understand the various layers of a Neural Network, and create a Deep Learning solution using Keras and Python.

We will cover the following topics in this chapter:

(1) What is tensor and how to use TensorFlow

(2) Demystifying Convolutional Neural Network

(3) Components of convolutional Neural Network

(4) Developing CNN network for image classification

2.1 Technical requirements

The code and datasets for the chapter are uploaded at the GitHub link https://github.com/Apress/computer-vision-using-deep-learning/tree/main/Chapter2 for this book. We will use the Jupyter Notebook. For this chapter, a CPU is good enough to execute the code, but if required you can use Google Colaboratory. You can refer to the reference at the end of the book, if you are not able to set up the Google Colab yourself.

2.2 Deep Learning using TensorFlow and Keras

Let us examine TensorFlow (TF) and Keras briefly now. They are arguably the most common open source libraries.

TensorFlow (TF) is a platform for Machine Learning by Google. *Keras* is a framework developed on top of other DL toolkits like TF, Theano, CNTK, and so on. It has built-in support for convolutional and recurrent Neural Networks.

Tip Keras is an API-driven solution; most of the heavy lifting is already done in Keras. It is easier to use and hence recommended for beginners.

The computations in TF are done using data flow graphs wherein the data is represented by edges (which are nothing but tensors or multidimensional data arrays) and nodes that represent mathematical operations. So, what exactly are tensors?

2.3 What is a tensor?

Recall *scalars* and *vectors* from your high-school mathematics. Vectors can be visualized as scalars with a direction. For example, a speed of 50 km/hr is a scalar, while 50 km/hr in the north direction is a vector. This means that a vector is a scalar magnitude in a given direction. A *tensor*, on the other hand, will be in multiple directions, that is, scalar magnitudes in multiple directions.

In terms of a mathematical definition, a tensor is an object that can provide a linear mapping between two algebraic objects. These objects can themselves be scalars or vectors or even tensors.

A tensor can be visualized in a vector space diagram as shown in Figure 2-1.

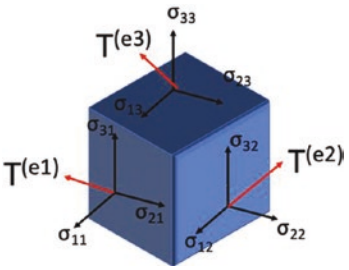

Figure 2-1. *A tensor represented in a vector space diagram. A tensor is a scalar magnitude in multiple directions and is used to provide linear mapping between two algebraic objects*

As you can see in Figure 2-1, a tensor has projections across multiple directions. A tensor can be thought of as a mathematical entity, which is described using components. They are described with reference to a basis, and if this associated basis changes, the tensor has to change itself. An example is coordinate change; if a transformation is done on the basis, the tensor's numeric value will also change. TensorFlow uses these tensors to make complex computations.

Now let's develop a basic check to see if you have installed TF correctly. We are going to multiply two constants to check if the installation is correct.

Info Refer to Chapter 1 if you want to know how to install TensorFlow and Keras.

---

1. Let's import TensorFlow:

   ```
   import tensorflow as tf
   ```

2. Initialize two constants:

   ```
   a = tf.constant([1,1,1,1])
   b = tf.constant([2,2,2,2])
   ```

3. Multiply the two constants:

```
product_results = tf.multiply(a, b)
```

4. Print the final result:

```
print(product_results)
```

If you are able to get the results, congratulations you are all set to go!

Now let us study a Convolutional Neural Network in detail. After that, you will be ready to create your first image classification model.

Exciting, right?

2.3.1 What is a Convolutional Neural Network?

When we humans see an image or a face, we are able to identify it immediately. It is one of the basic skills we have. This identification process is an amalgamation of a large number of small processes and coordination between various vital components of our visual system.

A Convolutional Neural Network or CNN is able to replicate this astounding capability using Deep Learning.

Consider this. We have to create a solution to distinguish between a cat and a dog. The attributes which make them different can be ears, whiskers, nose, and so on. CNNs are helpful for extracting the attributes of the images which are significant for the images. Or in other words, CNNs will extract the features which are distinguishing between a cat and a dog. CNNs are very powerful in image classification, object detection, object tracking, image captioning, face recognition, and so on.

Let us dive into the concepts of CNN. We will examine convolution first.

2.3.2 What is convolution?

The primary objective of the convolution process is to extract features which are important for image classification, object detection, and so on. The features will be edges, curves, color drops, lines and so on. Once the process has been trained well, it will learn these attributes at a significant point in the image. And then it can detect it later in any part of the image.

Imagine you have an image of size 32x32. That means it can be represented as 32x32x3 pixels if it is a colored image (remember RGB). Now let us move (or *convolute*) an area of 5x5 over this image and cover the entire image. This process is called *convolving*. Starting from the top left, this area is passed over the entire image. You can refer to Figure 2-2 to see how an image 32x32 is being convoluted by a filter of 5x5.

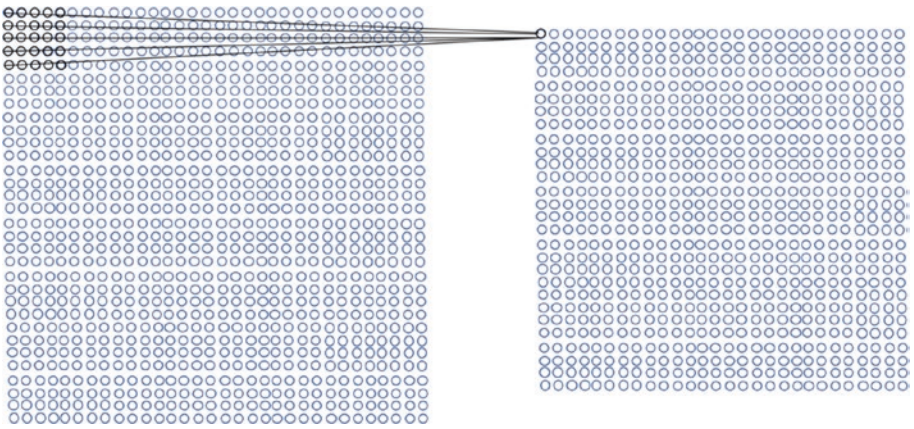

Figure 2-2. *Convolution process: input layer is on the left, output is on the right. The 32x32 image is being convoluted by a filter of size 5x5*

The 5x5 area which is passed over the entire image is called a *filter* which is sometimes called a *kernel* or *feature detector*. The region which is highlighted in Figure 2-2 is called the filter's *receptive field*. Hence, we can say that a filter is just a matrix with values called weights. These weights are trained and updated during the model training process. This filter moves over each and every part of the image.

We can understand the convolutional process by means of an example of the complete process as shown in Figure 2-3. The original image is 5x5 in size, and the filter is 3x3 in size. The filter moves over the entire image and keeps on generating the output.

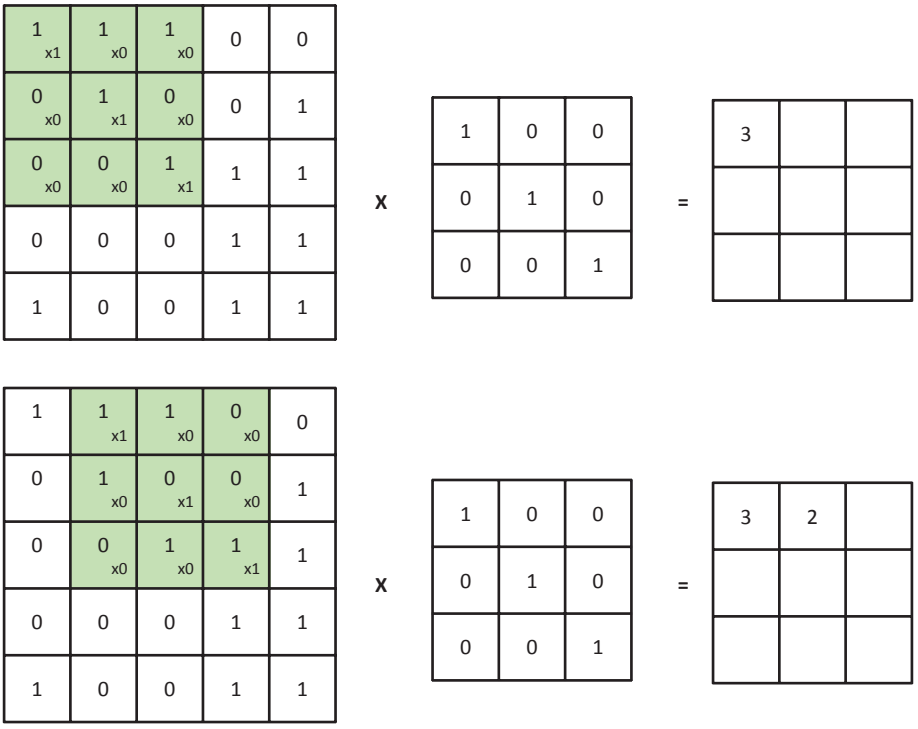

Figure 2-3. *Convolution is the process where the element-wise product and the addition are done. In the first image, the output is 3, and in the second image, the filter has shifted one place to the right and the output is 2*

In Figure 2-3, the 3x3 filter is convolving over the entire image. The filter checks if the feature it meant to detect is present or not. The filter carries a convolution process, which is the element-wise product and sum between the two metrics. If a feature is present, the convolution output of the filter and the part of the image will result in a high number. If the feature is not

present, the output will be low. Hence, this output value represents how confident a filter is that a particular feature is present in the image.

We move this filter over the entire image, resulting in an output matrix called feature maps or activation maps. This feature map will have the convolutions of the filter over the entire image.

Let's say the dimensions of the input image are (n,n) and the dimensions of filter are (x,x).

So, the output after the CNN layer is **((n-x+1), (n-x+1))**.

Hence, in the example in Figure 2-3, the output is (5-3+1, 5-3+1) = (3,3).

There is one more component called *channels* which is of much interest. Channels are the depth of matrices involved in the convolution process. The filter will be applied to each of the channels in the input image. We are again representing the output of the process in Figure 2-4. The input image is 5x5x5 in size, and we have a filter of 3x3x3. The output hence becomes an image of size (3x3). We should note that the filter should have exactly the same number of channels as the input image. In Figure 2-4, it is 3. It is to allow the element-wise multiplication between the metrics.

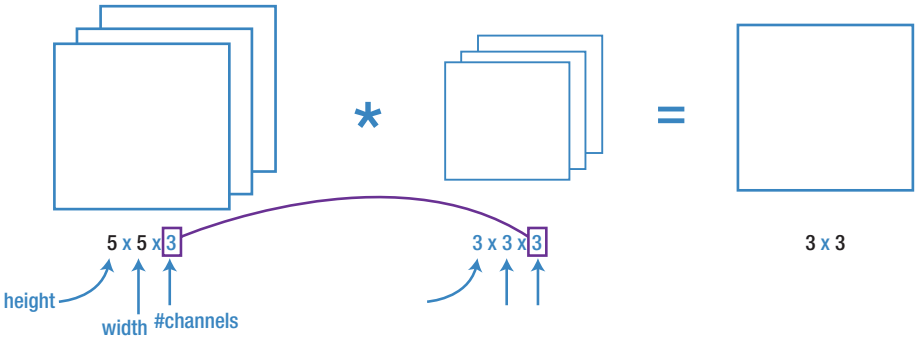

Figure 2-4. *The filter has the same number of channels as the input image*

There is one more component which we should be aware of. The filter can slide over the input image at varying intervals. It is referred to as the *stride value*, and it suggests how much the filter should move at each of the steps.

The process is shown in Figure 2-5. In the first figure on the left, we have a single stride, while on the right, a two-stride movement has been shown.

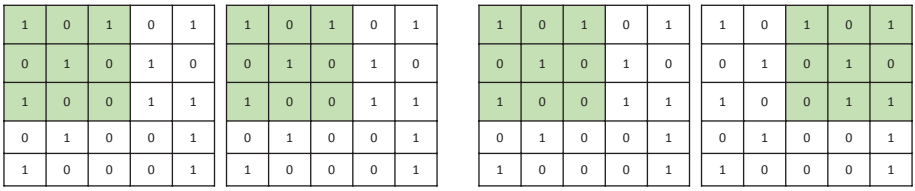

Figure 2-5. *Stride suggests how much the filter should move at each step. The figure shows the impact of a stride on convolution. In the first figure, we have a stride of 1, while in the second image, we have a stride of 2*

You would agree that with the convolution process, we will quickly lose the pixels along the periphery. As we have seen in Figure 2-3, a 5x5 image became a 3x3 image, and this loss will increase with a greater number of layers. To tackle this problem, we have the concept of padding. In padding, we add some pixels to an image which is getting processed. For example, we can pad the image with zeros as shown in Figure 2-6. The usage of padding results in a better analysis and better results from the convolutional Neural Networks.

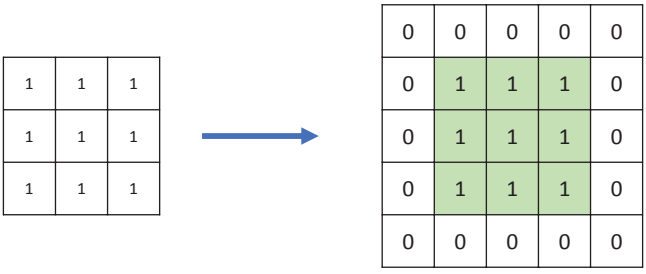

Figure 2-6. *Zero padding has been added to the input image. Convolution as the process reduces the number of pixels, padding allows us to tackle it*

Now we have understood the prime components of a CNN. Let us combine the concepts and create a small process. If we have an image of size (nxn) and we apply a filter of size "f," with a stride of "s" and padding "s," then the output of the process will be

$$((n+2p-f)/s+1), (n+2p-f)/s+1)) \qquad \text{(Equation 2-1)}$$

We can understand the process as shown in Figure 2-7. We have an input image of size 37x37 and a filter of size 3x3, and the number of filters is 10, the stride is 1, and the padding is 0. Based on Equation 2-1, the output will be 35x35x10.

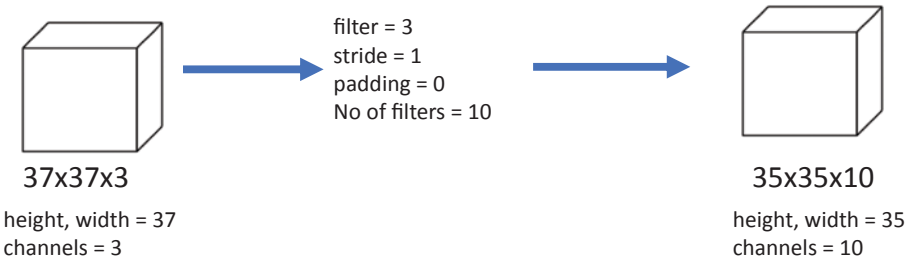

Figure 2-7. *A convolution process in which we have a filter of size 3x3, stride of 1, padding of 0, and number of filters as 10*

Convolution helps us in extracting the significant attributes of the image. The layers of the network closer to the origin (the input image) learn the low-level features, while the final layers learn the higher features. In the initial layers of the network, features like edges, curves, and so on are getting extracted, while the deeper layers will learn about the resulting shapes from these low-level features like face, object, and so on.

But this computation looks complex, isn't it? And as the network will go deep, this complexity will increase. So how do we deal with it? Pooling layer is the answer. Let's understand it.

2.3.3 What is a Pooling Layer?

The CNN layer which we studied results in a feature map of the input. But as the network becomes deeper, this computation becomes complex. It is due to the reason that with each layer and neuron, the number of dimensions in the network increases. And hence the overall complexity of the network increases.

And, there's one more challenge: any image augmentation will change the feature map. For example, a rotation will change the position of a feature in the image, and hence the respective feature map will also change.

Note Often, you will face nonavailability of raw data. Image augmentation is one of the recommended methods to create new images for you which can serve as training data.

This change in the feature map can be addressed by *downsampling*. In downsampling, a lower resolution of the input image is created, and the *Pooling Layer* helps us with this.

A Pooling Layer is added after the Convolutional Layer. Each of the feature maps is operated upon individually, and we get a new set of pooled feature maps. The size of this operation filter is smaller than the feature map's size.

A pooling layer is generally applied after the convolutional layer. A pooling layer with 3x3 pixels and a stride diminishes feature maps' size by a factor of 2 which means that each dimension is halved. For example, if we apply a pooling layer to a feature map of 8x8 (64 pixels), the output will be a feature map of 4x4 (16 pixels).

There are two types of Pooling Layers.

Average Pooling and Max Pooling. The former calculates the average of each value of the feature map, while the latter gets the maximum value for each patch of the feature map. Let us examine them as shown in Figure 2-8.

Figure 2-8. *The right figure is the max pooling, while the bottom is the average pooling*

As you can see in Figure 2-8, the average pooling layer does an average of the four numbers, while the max pooling selects the maximum from the four numbers.

There is one more important concept about Fully Connected layers which you should know before you are equipped to create a CNN model. Let us examine it and then you are good to go.

2.3.4 What is a Fully Connected Layer?

A Fully Connected layer takes input from the outputs of the preceding layer (activation maps of high-level features) and outputs an n-dimensional vector. Here, n is the number of distinct classes.

For example, if the objective is to ascertain if an image is of a horse, a fully connected layer will have high-level features like tail, legs, and so on in the activation maps. A fully connected layer looks like Figure 2-9.

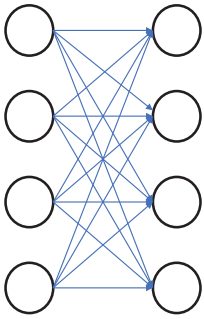

Figure 2-9. *A fully connected layer is depicted here*

A Fully Connected layer will look at the features which correspond most closely to a particular class and have particular weights. This is done to get correct probabilities for different classes when we get the product between the weights and the previous layer.

Now you have understood CNN and its components. It is time to hit the code. You will create your first Deep Learning solution to classify between cats and dogs. All the very best!

2.4 Developing a DL solution using CNN

We will now create a Deep Learning solution using CNN. The "Hello World" of Deep Learning is generally called an MNIST dataset. It is a collection of handwritten numerical digits as depicted in Figure 2-10.

Figure 2-10. *MNIST dataset: "Hello World" for image recognition*

There is a famous paper on recognizing MNIST images (description is given at the end of the chapter). To avoid repetition, we are uploading the entire code at GitHub. You are advised to check the code.

We will now start creating the image classification model!

In this first Deep Learning solution, we want to distinguish between a cat and a dog based on their image. The dataset is available at `www.kaggle.com/c/dogs-vs-cats`.

The steps to be followed are listed here:

1. First, let's build the dataset:
    a. Download the dataset from Kaggle. Unzip the dataset.
    b. You will find two folders: test and train. Delete the test folder as we will create our own test folder.
    c. Inside both the train and test folders, create two subfolders – cats and dogs – and put the images in the respective folders.

d. Take some images (I took 2000) from the "train>cats" folder and put them in the "test>cats" folder.

e. Take some images (I took 2000) from the "train>dogs" folder and put them in the "test>dogs" folder.

f. Your dataset is ready to be used.

2. Import the required libraries now. We will import `sequential`, `pooling`, `activation`, and `flatten` layers from `keras`. Import `numpy` too.

Note In the Reference of the book, we provide the description of each of the layers and their respective functions.

```
from keras.models import Sequential
from keras.layers import Conv2D,Activation,
MaxPooling2D,Dense,Flatten,Dropout
import numpy as np
```

3. Initialize a model, `catDogImageclassifier` variable here:

```
catDogImageclassifier = Sequential()
```

4. Now, we'll add layers to our network. `Conv2D` will add a two-dimensional convolutional layer which will have 32 filters. 3,3 represents the size of the filter (3 rows, 3 columns). The following input image shape is 64*64*3 – height*width*RGB. Each number represents the pixel intensity (0–255).

```
catDogImageclassifier.add(Conv2D(32,(3,3),
input_shape=(64,64,3)))
```

5. The output of the last layer will be a feature map. The training data will work on it and get some feature maps.

6. Let's add the activation function. We are using `ReLU` (Rectified Linear Unit) for this example. In the feature map output from the previous layer, the activation function will replace all the negative pixels with zero.

Note Recall from the definition of ReLU; it is max(0,x). ReLU allows positive values while replaces negative values with 0. Generally, ReLU is used only in hidden layers.

```
catDogImageclassifier.add(Activation('relu'))
```

7. Now we add the Max Pooling layer as we do not want our network to be overly complex computationally.

```
catDogImageclassifier.add(MaxPooling2D
(pool_size =(2,2)))
```

8. Next, we add all three convolutional blocks. Each block has a Cov2D, ReLU, and Max Pooling Layer.

```
catDogImageclassifier.add(Conv2D(32,(3,3)))
catDogImageclassifier.add(Activation('relu'))
catDogImageclassifier.add(MaxPooling2D(pool_size =(2,2)))
catDogImageclassifier.add(Conv2D(32,(3,3)))
catDogImageclassifier.add(Activation('relu'))
catDogImageclassifier.add(MaxPooling2D(pool_size =(2,2)))
```

```
catDogImageclassifier.add(Conv2D(32,(3,3
catDogImageclassifier.add(Activation('relu'))
catDogImageclassifier.add(MaxPooling2D(pool_size
=(2,2)))
```

9. Now, let's flatten the dataset which will transform the pooled feature map matrix into one column.

```
catDogImageclassifier.add(Flatten())
```

10. Add the dense function now followed by the ReLU activation:

```
catDogImageclassifier.add(Dense(64))
catDogImageclassifier.add(Activation('relu'))
```

Info Do you think why we need nonlinear functions like tanh, ReLU, and so on? If you use only linear functions, the output will be linear too. Hence, we use nonlinear functions in hidden layers.

---

11. Overfitting is a nuisance. We will add the Dropout layer to overcome overfitting next:

```
catDogImageclassifier.add(Dropout(0.5))
```

12. Add one more fully connected layer to get the output in n-dimensional classes (a vector will be the output).

```
catDogImageclassifier.add(Dense(1))
```

13. Add the Sigmoid function to convert to probabilities:

```
catDogImageclassifier.add(Activation('sigmoid'))
```

14. Let's print a summary of the network.

```
catDogImageclassifier.summary()
```

We see the entire network in the following image:

```
1 catDogImageclassifier.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_27 (Conv2D)           (None, 62, 62, 32)        896
_________________________________________________________________
activation_22 (Activation)   (None, 62, 62, 32)        0
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 29, 29, 32)        9248
_________________________________________________________________
activation_23 (Activation)   (None, 29, 29, 32)        0
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 12, 12, 32)        9248
_________________________________________________________________
activation_24 (Activation)   (None, 12, 12, 32)        0
_________________________________________________________________
max_pooling2d_22 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 4, 4, 32)          9248
_________________________________________________________________
activation_25 (Activation)   (None, 4, 4, 32)          0
_________________________________________________________________
max_pooling2d_23 (MaxPooling (None, 2, 2, 32)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
activation_26 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
_________________________________________________________________
activation_27 (Activation)   (None, 1)                 0
=================================================================
Total params: 36,961
Trainable params: 36,961
Non-trainable params: 0
```

We can see the number of parameters in our network is 36,961. You are advised to play around with different network structures and gauge the impact.

15. Let us now compile the network. We use the optimizer `rmsprop` using gradient descent, and then we add the loss or the cost function.

```
catDogImageclassifier.compile(optimizer =
'rmsprop', loss ='binary_crossentropy',
metrics =['accuracy'])
```

16. Now we are doing data augmentation here (zoom, scale, etc.). It will also help to tackle the problem of overfitting. We use the `ImageDataGenerator` function to do this:

```
from keras.preprocessing.image import
ImageDataGenerator
train_datagen = ImageDataGenerator(rescale
=1./255, shear_range =0.25,zoom_range = 0.25,
horizontal_flip =True)
test_datagen = ImageDataGenerator(rescale =
1./255)
```

17. Load the training data:

```
training_set = train_datagen.flow_from_directory
('/Users/DogsCats/train',target_size=(64,6 4),
batch_size= 32,class_mode='binary')
```

18. Load the testing data:

```
test_set = test_datagen.flow_from_directory
('/Users/DogsCats/test', target_size = (64,64),
batch_size = 32,
class_mode ='binary')
```

19. Let us begin the training now.

```
from IPython.display import display
from PIL import Image catDogImageclassifier.fit_
generator(training_set, steps_per_epoch =625,
epochs = 10, validation_data =test_set,
validation_steps = 1000)
```

Steps per epoch are 625, and the number of epochs is 10. If we have 1000 images and a batch size of 10, the number of steps required will be 100 (1000/10).

Info The number of *epochs* means the number of complete passes through the full training dataset. *Batch* is the number of training examples in a batch, while *iteration* is the number of batches needed to complete an epoch.

Depending on the complexity of the network, the number of epochs given, and so on, the compilation will take time. The test dataset is passed as a `validation_data` here.

The output is shown in Figure 2-11.

```
from IPython.display import display
from PIL import Image
catDogImageclassifier.fit_generator(training_set,
                         steps_per_epoch =625,
                         epochs = 10,
                         validation_data =test_set,
                         validation_steps = 1000)
```

```
Epoch 1/10
625/625 [==============================] - 185s 296ms/step - loss: 0.6721 - acc: 0.5822 - val_loss: 0.6069 - val_acc:
0.6610
Epoch 2/10
625/625 [==============================] - 152s 243ms/step - loss: 0.5960 - acc: 0.6831 - val_loss: 0.5151 - val_acc:
0.7543
Epoch 3/10
625/625 [==============================] - 151s 242ms/step - loss: 0.5452 - acc: 0.7217 - val_loss: 0.4891 - val_acc:
0.7545
Epoch 4/10
625/625 [==============================] - 150s 239ms/step - loss: 0.5069 - acc: 0.7568 - val_loss: 0.4657 - val_acc:
0.7743
Epoch 5/10
625/625 [==============================] - 150s 240ms/step - loss: 0.4813 - acc: 0.7713 - val_loss: 0.4407 - val_acc:
0.7925
Epoch 6/10
625/625 [==============================] - 152s 243ms/step - loss: 0.4526 - acc: 0.7866 - val_loss: 0.4374 - val_acc:
0.7924
Epoch 7/10
625/625 [==============================] - 151s 241ms/step - loss: 0.4458 - acc: 0.7953 - val_loss: 0.3891 - val_acc:
0.8324
Epoch 8/10
625/625 [==============================] - 151s 242ms/step - loss: 0.4177 - acc: 0.8123 - val_loss: 0.3917 - val_acc:
0.8221
Epoch 9/10
625/625 [==============================] - 155s 248ms/step - loss: 0.4158 - acc: 0.8158 - val_loss: 0.3947 - val_acc:
0.8176
Epoch 10/10
625/625 [==============================] - 151s 241ms/step - loss: 0.4021 - acc: 0.8201 - val_loss: 0.3783 - val_acc:
0.8221

<keras.callbacks.History at 0x1a376b5160>
```

Figure 2-11. *Output of the training results for 10 epochs*

As seen in the results, in the final epoch, we got validation accuracy of 82.21%. We can also see that in Epoch 7 we got an accuracy of 83.24% which is better than the final accuracy.

We would then want to use the model created in Epoch 7 as the accuracy is best for it. We can achieve it by providing checkpoints between the training and saving that version. We will look at the process of creating and saving checkpoints in subsequent chapters.

We have saved the final model as a file here. The model can then be loaded again as and when required. The model will be saved as an HDF5 file, and it can be reused later.

```
catDogImageclassifier.save('catdog_cnn_model.h5')
```

20. Load the saved model using load_model:

```
from keras.models import load_model
catDogImageclassifier = load_model('catdog_
cnn_model.h5')
```

21. Check how the model is predicting an unseen image.

Here's the image (Figure 2-12) we are using to make a prediction. Feel free to test the solution with different images.

Figure 2-12. *A dog image to test the accuracy of the model*

In the following code block, we make a prediction on the preceding image using the model we have trained.

22. Load the library and the image from the folder. You would have to change the location of the file in code snippet below.

```
import numpy as np
from keras.preprocessing import image
an_image =image.load_img('/Users/vaibhavverdhan/Book
Writing/2.jpg',target_size =(64,64))
```

The image is now getting converted to an array of numbers:

```
an_image =image.img_to_array(an_image)
```

Let us now expand the image's dimensions. It will improve the prediction power of the model. It is used to expand the shape of an array and inserts a new axis which appears at the position of the axis in the expanded array shape.

```
an_image =np.expand_dims(an_image, axis =0)
```

It is time to call the predict method now. We are setting the probability threshold at 0.5. You are advised to test on multiple thresholds and check the corresponding accuracy.

```
verdict = catDogImageclassifier.predict(an_image) if verdict[0]
[0] >= 0.5:
prediction = 'dog'
else:
prediction = 'cat'
```

23. Let us print our final prediction.

```
print(prediction)
```

The model predicts that the image is of a "dog."

Here, in this example, we designed a Neural Network using Keras. We trained the images using images of cats and dogs and tested it. It is possible to train a multiclassifier system too if we can get the images for each of the classes.

Congratulations! You just now finished your second image classification use case using Deep Learning. Use it for training your own image datasets. It is even possible to create a multiclass classification model.

Now you may think how you will use this model to make predictions in real time. The compiled model file (e.g., `'catdog_cnn_model.h5'`) will be deployed onto a server to make the predictions. We will be covering model deployment in detail in the last chapter of this book.

With this, we come to the close of the second chapter. You can proceed to the summary now.

2.5 Summary

Images are a rich source of information and knowledge. We can solve a lot of business problems by analyzing the image dataset. CNNs are leading the AI revolution particularly for images and videos. They are being used in the medical industry, manufacturing, retail, BFSI, and so on. Quite a few researches are going on using CNN.

CNN-based solutions are quite innovative and unique. There are a lot of challenges which have to be tackled while designing an innovative CNN-based solution. The choice of the number of layers, number of neurons in each layer, activation functions to be used, loss function, optimizer, and so on is not a straightforward one. It depends on the complexity of the business problem, the dataset at hand, and the available computation power. The efficacy of the solution depends a lot on the dataset available. If we have a clearly defined business objective which is measurable, precise, and achievable, if we have a representative and complete dataset, and if we have enough computation power, a lot of business problems can be addressed using Deep Learning.

In the first chapter of the book, we introduced computer vision and Deep Learning. In this second chapter, we studied concepts of convolutional, pooling, and fully connected layers. You are going to use these concepts throughout your journey. And you also developed an image classification model using Deep Learning.

The difficulty level is going to increase from the next chapter onward when we start with network architectures. Network architectures use the building blocks we have studied in the first two chapters. They are developed by scientists and researchers across organizations and universities to solve complex problems. We are going to study those networks and develop Python solutions too. So stay hungry!

You should be able to answer the questions in the exercise now!

REVIEW QUESTIONS

You are advised to solve these questions:

1. What is the convolution process in CNN and how is the output calculated?
2. Why do we need nonlinear functions in hidden layers?
3. What is the difference between max and average pooling?
4. What do you mean by dropout?
5. Download the image data of natural scenes around the world from `www.kaggle.com/puneet6060/intel-image-classification` and develop an image classification model using CNN.
6. Download the Fashion MNIST dataset from `https://github.com/zalandoresearch/fashion-mnist` and develop an image classification model.

2.5.1 Further readings

You are advised to go through the following papers:

1. Go through "Assessing Four Neural Networks on Handwritten Digit Recognition Dataset (MNIST)" at https://arxiv.org/pdf/1811.08278.pdf.
2. Study the research paper "A Survey of the Recent Architectures of Deep Convolutional Neural Networks" at https://arxiv.org/pdf/1901.06032.pdf.
3. Go through the research paper "Understanding Convolutional Neural Networks with a Mathematical Model" at https://arxiv.org/pdf/1609.04112.pdf.
4. Go through "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition" at http://ais.uni-bonn.de/papers/icann2010_maxpool.pdf.