LITTORAL CÔTE D'OPALE UNIVERSITY
ENGINEERING SCHOOL OF ULCO

Project : Image Classification Using Convolutions Neural Network

Author:
Oumaima Aabi

Supervisor:
Pr. Khalide Jbilou

Second year of master's degree
Engineering of Complex Systems
(MIASC M2)

in the Department of Mathematics

December 23, 2021

Abstract

In recent year, with the speedy development in the digital contents identification, automatic classification of the images became most challenging task in the fields of computer vision. Automatic understanding and analysing of images by system is difficult as compared to human visions. Several research have been done to overcome problem in existing classification system, but the output was narrowed only to low level image primitives. However, those approach lack with accurate classification of images. In this paper, our system uses deep learning algorithm to achieve the expected results in the area like computer visions. Our system present Convolutional Neural Network (CNN), a machine learning algorithm being used for automatic classification the images.

We will approach the subject in the following way:

1. In Chapter 1, we will present state-of-the-art Convolutional Neural Networks.
2. In Chapter 2, we will study some of the fundamentals of Convolutional Neural Networks.
3. In Chapter 3, we will jump to the implementation of Convolutional Neural Networks for image classification problems, and we will end this report with a conclusion.

The document was typeset in LaTeX. In this project we will be using a dataset named **CIFAR-10** dataset. The system has been implemented by Python Jupyter Notebook, and the source codes are attached to the appendix in the end of this report.

Please don't hesitate to contact me for any remark or error.

My e-mail: oumaima.aabi@etu.univ-littoral.fr
My advisor's e-mail: khalide.jbilou@univ-littoral.fr

Keywords: Deep learning, Convolutional Neural Network (CNN), Image Classification, CIFAR-10 Image Datasets, Computer Vision.

Contents

List of Figures

Chapter 1

The state-of-the-art

1.1 Introduction

1.1.1 Image Classification

In recent years, due to the explosive growth of digital content, automatic classification of images has become one of the most critical challenges in visual information indexing and retrieval systems.Computer vision is an interdisciplinary and subfield of artificial intelligence that aims to give similar capability of human to computer for understanding information from the images. Several research efforts were made to overcome these problems, but these methods consider the low-level features of image primitives. Focusing on low-level image features will not help to process the images.

Image classification is a big problem in computer vision for the decades. In case of humans the image understanding, and classification is done very easy task, but in case of computers it is very expensive task. In general, each image is composed of set of pixels and each pixel is represented with different values. Henceforth to store an image the computer must need more spaces for store data. To classify images, it must perform higher number of calculations. For this it requires systems with higher configuration and more computing power. In real time to take decisions basing on the input is not possible because it takes more time for performing these many computations to provide result.

1.1.2 Convolutional neural networks (CNN) in Computer Vision

Deep convolutional neural networks have recently been substantially improving upon the state of the art in image classification and other recognition tasks. Since their introduction in the early 1990s, convolutional neural networks **CNN** have consistently been competitive with other techniques for image classification and recognition.

The most common use for **CNNs** is image classification, for example identifying satellite images that contain roads or classifying hand written letters and digits. There are other quite mainstream tasks such as image segmentation and signal processing, for which **CNNs** perform well at (2.1).

Recently, they have pulled away from competing methods due the availability of larger data sets, better models and training algorithms and the availability of GPU computing to enable investigation of larger and deeper models.

CNNs have been used for understanding in Natural Language Processing (NLP) and speech recognition, although often for NLP Recurrent Neural Nets (RNNs) are used.

A **CNN** can also be implemented as a U-Net architecture, which are essentially two almost mirrored **CNNs** resulting in a **CNN** whose architecture can be presented in a U shape. U-nets are used where the output needs to be of similar size to the input such as segmentation and image improvement.

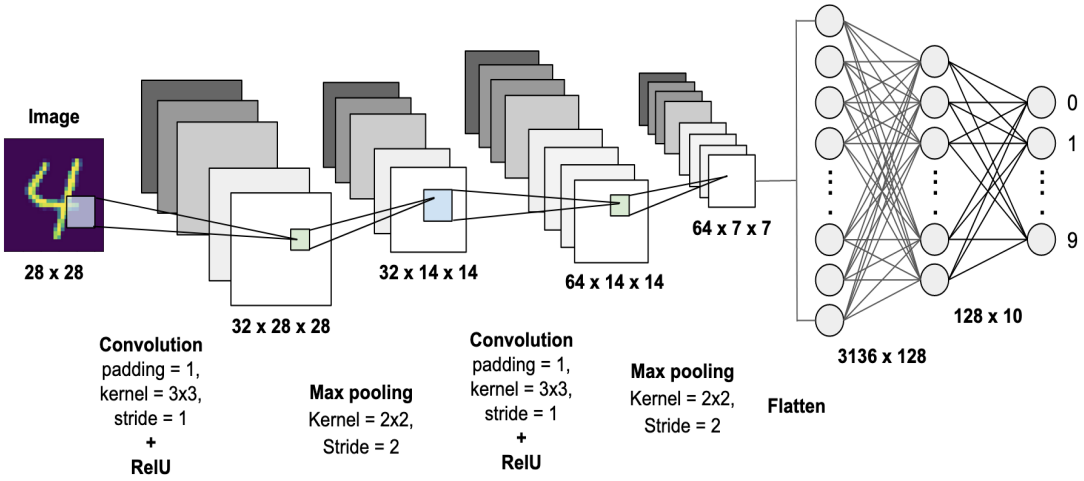

FIGURE 1.1: Example of CNN on Handwritten Digits (MNIST)

Chapter 2

Fundamentals of Convolutional Neural Networks

2.1 Neurons in Human Vision

A convolutional neural network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex 2.1. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

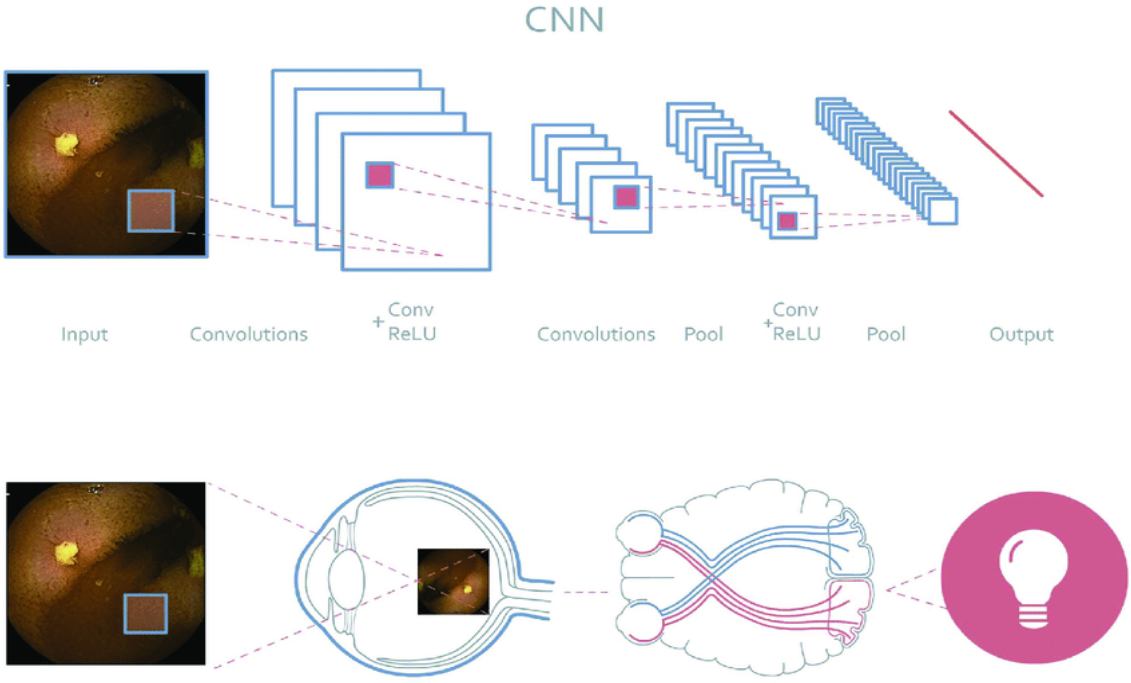

FIGURE 2.1: Schematic representation of the analogy between a CNN and a biologic visual cortical pathway.

2.2 Convolutional Neural Networks (CNN)

The convolutional network takes advantage of the fact that we're analyzing images, and sensibly constrains the architecture of the deep network so that we drastically reduce the number of parameters in our model. Inspired by how human vision works, layers of a convolutional network have neurons arranged in three dimensions, so layers have a width, height, and depth, as shown in Figure 2.2.

As we'll see, the neurons in a convolutional layer are only connected to a small, local region of the preceding layer, so we avoid the wastefulness of fully-connected neurons. A convolu- tional layer's function can be expressed simply: it processes a three-dimensional vol- ume of information to produce a new three-dimensional volume of information. We'll take a closer look at how this works in the next section.

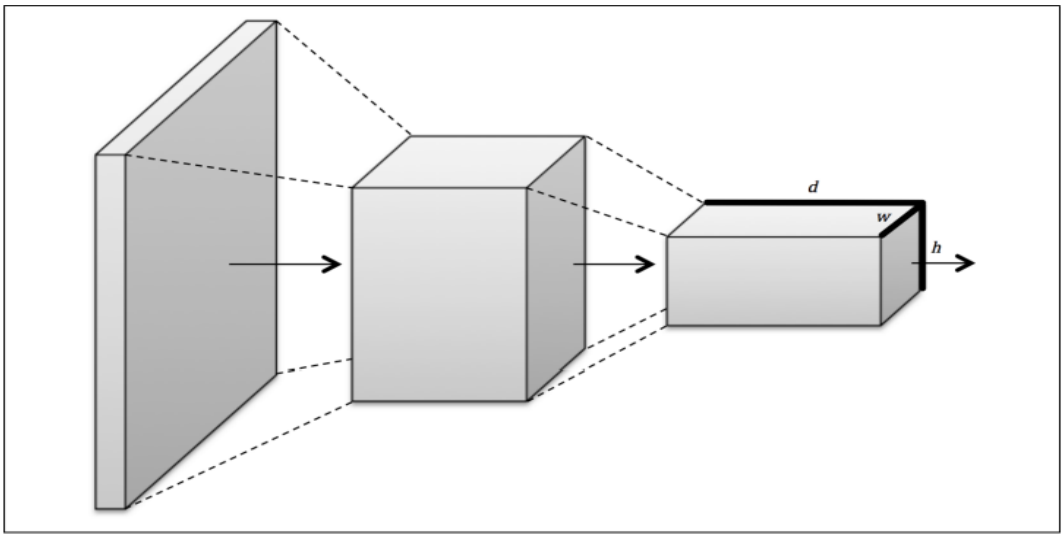

FIGURE 2.2: Convolutional layers arrange neurons in three dimensions, so layers have width, height, and depth.

2.2.1 Filters and Feature Maps

In order to motivate the primitives of the convolutional layer, let's build an intuition for how the human brain pieces together raw visual information into an understand- ing of the world around us. One of the most influential studies in this space came from *David Hubel* and *Torsten Wiesel*, who discovered that parts of the visual cortex are responsible for detecting edges. In 1959, they inserted electrodes into the brain of a cat and projected black-and-white patterns on the screen. They found that some neurons fired only when there were vertical lines, others when there were horizontal lines, and still others when the lines were at particular angles.

Further work determined that the visual cortex was organized in layers. Each layer is responsible for building on the features detected in the previous layers—from lines, to contours, to shapes, to entire objects. Furthermore, within a layer of the visual cortex, the same feature detectors were replicated over the whole area in order to detect features in all parts of an image.These ideas significantly impacted the design of convolutional neural nets.

The first concept that arose was that of a **filter** (also called kernel). A filter is essentially a feature detector, and to understand how it works, let's consider the toy image in Figure 2.3.

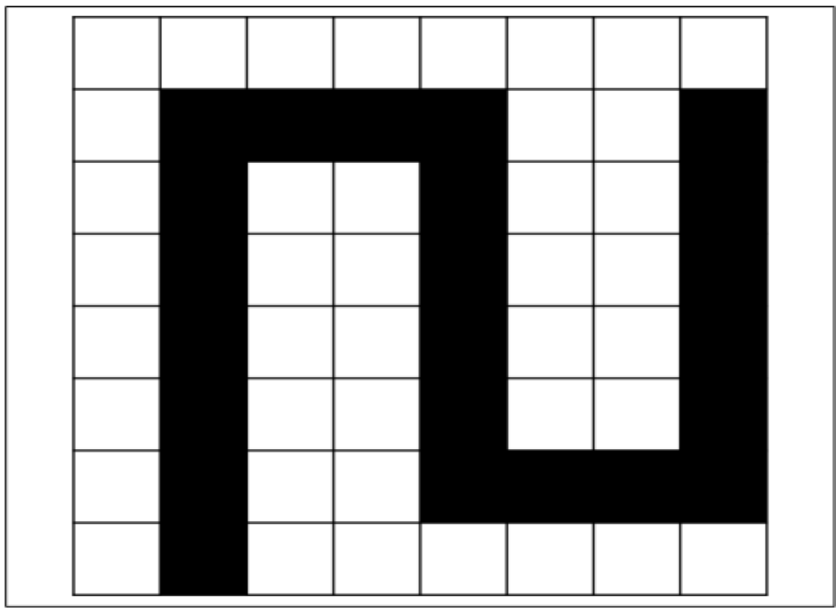

FIGURE 2.3: We'll analyze this simple black-and-white image as a toy example.

Let's say that we want to detect vertical and horizontal lines in the image. One approach would be to use an appropriate feature detector, as shown in. For example, to detect vertical lines, we would use the feature detector on the top, slide it across the entirety of the image, and at every step check if we have a match. We keep track of our answers in the matrix in the top right. If there's a match, we shade the appropriate box black. If there isn't, we leave it white. This result is our feature map, and it indicates where we've found the feature we're looking for in the original image.
We can do the same for the horizontal line detector (bottom), resulting in the feature map in the bottom-right corner.

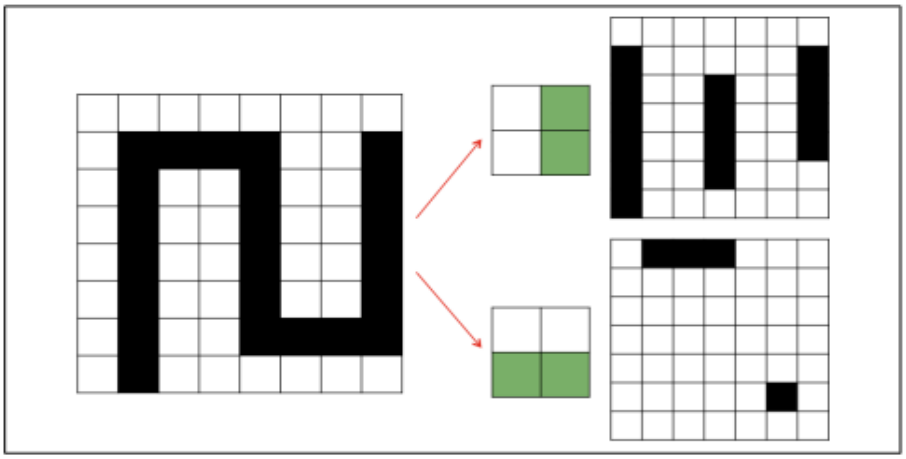

FIGURE 2.4: Applying filters that detect vertical and horizontal lines on our toy example.

This operation is called a **convolution**. We take a filter and we multiply it over the entire area of an input image.

Let's define four different filters and let's check later in the chapter their effect when used in convolution operations. For those examples, we will work with 3×3 filters.

- The following filter will allow the detection of horizontal edges

$$F_H = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

- The following filter will allow the detection of vertical edges

$$F_V = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

- The following filter will allow the detection of edges when luminosity changes drastically

$$F_L = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

- The following kernel will blur edges in an image

$$F_B = -\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

In the next sections, we will apply convolution to a test image with the filters, to see what their effect is.

2.2.2 Convolution

The first step to understanding CNNs is to understand convolution. The easiest way is to see it in action with a few simple cases. First, in the context of neural networks, convolution is done between tensors. The operation gets two tensors as input and produces a tensor as output. The operation is usually indicated with the operator $*$.

Let's see how it works. Consider two tensors, both with dimensions 3×3. The convolution operation is done by applying the following formula:

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} * \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix} = \sum_{i=1}^{9} a_i k_i$$

In this case, the result is merely the sum of each element, a_i, multiplied by the respective element, k_i. In more typical matrix formalism, this formula could be written

with a double sum as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} k_{ij}$$

However, the first version has the advantage of making the fundamental idea very clear: each element from one tensor is multiplied by the correspondent element (the element in the same position) of the second tensor, and then all the values are summed to get the result.

In the previous section, we talked about filters, and the reason is that convolution is usually done between a tensor, that we may indicate here with A, and a filter. Typically, filters are small, 3×3 or 5×5, while the input tensors A are normally bigger. In image recognition for example, the input tensors A are the images that may have dimensions as high as $1024 \times 1024 \times 3$, where 1024×1024 is the resolution and the last dimension (3) is the number of the color channels, the RGB values.

Example:

In advanced applications, the images may even have higher resolution. To understand how to apply convolution when we have matrices with different dimensions, let's consider a matrix A that is 4×4

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

And a filter K that we will take for this example to be 3×3

$$K = \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix}$$

The idea is to start in the top-left corner of the matrix A and select a 3×3 region. In the example that would be

$$A_1 = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \end{pmatrix}$$

Alternatively, the elements marked in boldface here:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Then we perform the convolution, as explained at the beginning between this smaller matrix A1 and K, getting (we will indicate the result with B1):

$$B_1 = A_1 * K = a_1 k_1 + a_2 k_2 + a_3 k_3 + k_4 a_5 + k_5 a_5 + k_6 a_7 + k_7 a_9 + k_8 a_{10} + k_9 a_{11}$$

Then we need to shift the selected 3 × 3 region in matrix A of one column to the right and select the elements marked in bold here:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

This will give us the second sub-matrix A2:

$$A_2 = \begin{pmatrix} a_2 & a_3 & a_4 \\ a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \end{pmatrix}$$

We then perform the convolution between this smaller matrix A2 and K again:

$$B_2 = A_2 * K = a_2k_1 + a_3k_2 + a_4k_3 + a_6k_4 + a_7k_5 + a_8k_6 + a_{10}k_7 + a_{11}k_8 + a_{12}k_9$$

We cannot shift our 3 × 3 region anymore to the right, since we have reached the end of the matrix A, so what we do is shift it one row down and start again from the left side. The next selected region would be

$$A_4 = \begin{pmatrix} a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \\ a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Moreover, the convolution will give this result:

$$B_4 = A_4 * K = a_6k_1 + a_7k_2 + a_8k_3 + a_{10}k_4 + a_{11}k_5 + a_{12}k_6 + a_{14}k_7 + a_{15}k_8 + a_{16}k_9$$

Now we cannot shift our 3 × 3 region anymore, neither right nor down. We have calculated four values: B1, B2, B3, and B4. Those elements will form the resulting tensor of the convolution operation giving us the tensor B:

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

The same process can be applied when tensor A is bigger. You will simply get a bigger resulting B tensor, but the algorithm to get the elements Bi is the same. Before moving on, there is still a small detail that we need to discuss, and that is the concept of stride. In the previous process, we moved our 3 × 3 region always one column to the right and one row down. The number of rows and columns, in this example 1, is called the stride and is often indicated with s. Stride s = 2 means simply that we shift our3 × 3 region two columns to the right and two rows down at each step 2.5.

Something else that we need to discuss is the size of the selected region in the input matrix A. The dimensions of the selected region that we shifted around in the process must be the same as of the kernel used. If you use a 5 × 5 kernel, you will need to select a 5 × 5 region in A. In general, given a $n_K \times n_K$ kernel, you select a $n_K \times n_K$ region in A.

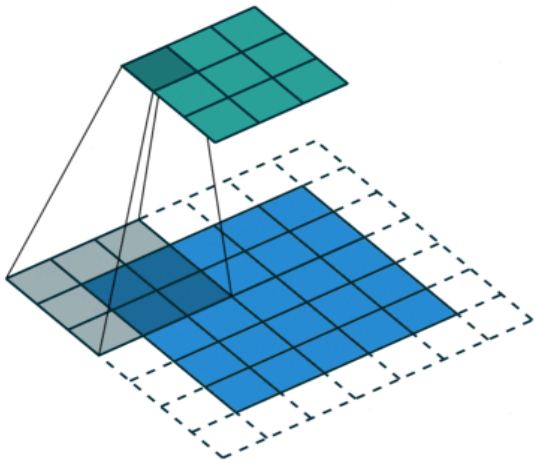

FIGURE 2.5: Convolution Operation with Stride $s = 2$.

In a more formal definition, convolution with stride s in the neural network context is a process that takes a tensor A of dimensions $n_A \times n_A$ and a kernel K of dimensions $n_K \times n_K$ and gives as output a matrix B of dimensions $n_B \times n_B$ with

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor$$

In Figure 2.6, you can see a visual explanation of how convolution works. Suppose to have a 3×3 filter. Then in the Figure 2.6, you can see that the top left nine elements of the matrix A, marked by a square drawn with a black continuous line, are the one used to generate the first element of the matrix B_1 according to this formula. The elements marked by the square drawn with a dotted line are the ones used to generate the second element B_2 and so on.

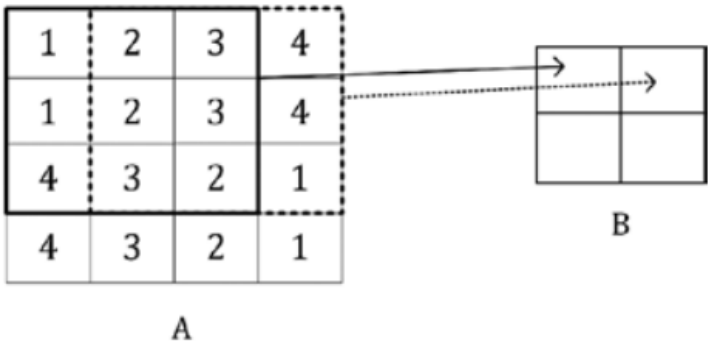

FIGURE 2.6: A visual explanation of convolution.

To reiterate what we discuss in the example at the beginning, the basic idea is that each element of the 3×3 square from matrix A is multiplied by the corresponding element of the kernel K and all the numbers are summed. The sum is then the element of the new matrix B. After having calculated the value for B_1, you shift the region you are considering in the original matrix of one column to the right (the square indicated in Figure 2.6 with a dotted line) and repeat the operation.

You continue to shift your region to the right until you reach the border and then you move one element down and start again from the left. You continue in this fashion until the lower right angle of the matrix. The same kernel is used for all the regions in the original matrix.

Given the kernel F_H for example, you can see in Figure 2.7 which element of A are multiplied by which elements in F_H and the result for the element B_1, that is nothing else as the sum of all the multiplications:

$$B_{11} = 1 \times 1 + 2 \times 1 + 3 \times 1 + 1 \times 0 + 2 \times 0 + 3 \times 0 + 4 \times (-1) + 3 \times (-1) + 2 \times (-1) = -3$$

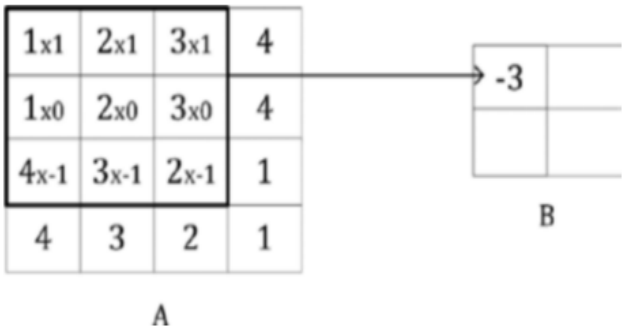

FIGURE 2.7: A visualization of convolution with the kernel F_H.

In Figure 2.8, you can see an example of convolution with stride $s = 2$.

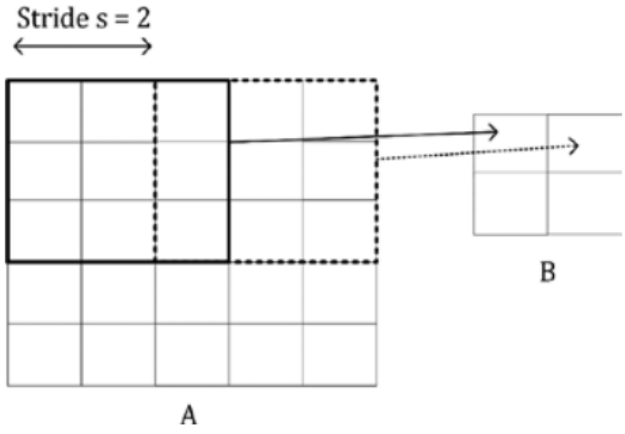

FIGURE 2.8: A visual explanation of convolution with stride $s = 2$.

2.2.3 Pooling

To aggressively reduce dimensionality of feature maps and sharpen the located features, we sometimes insert a max pooling layer after a convolutional layer.10 The essential idea behind max pooling is to break up each feature map into equally sized tiles. Then we create a condensed feature map. Specifically, we create a cell for each tile, compute the maximum value in the tile, and propagate this maximum value

into the corresponding cell of the condensed feature map. This process is illustrated in 2.11.

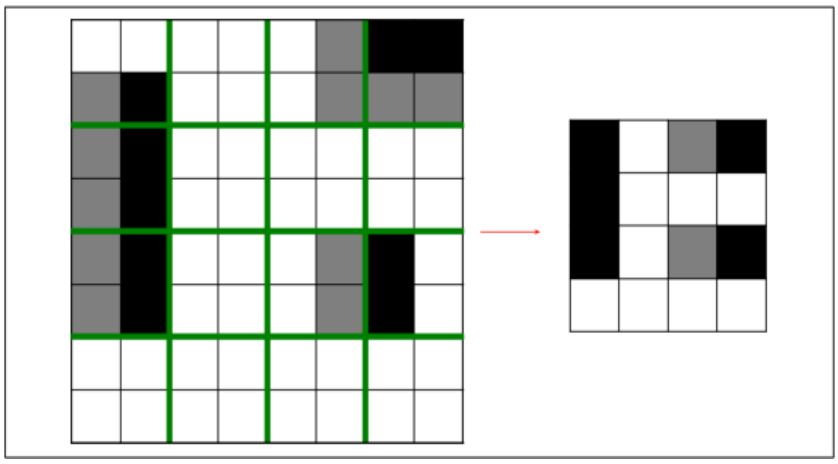

FIGURE 2.9: An illustration of how max pooling significantly reduces parameters as we move up the network.

To understand it, let's look at a concrete example and consider what is called max pooling. Consider the 4 × 4 matrix we discussed during our convolution discussion again:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

To perform max pooling, we need to define a region of size $n_K n_K$, analogous to what we did for convolution. Let's consider $n_K = 2$. What we need to do is start on the top-left corner of our matrix A and select a $n_K n_K$ region, in our case 2 × 2 from A. Here we would select

$$\begin{pmatrix} a_1 & a_2 \\ a_5 & a_6 \end{pmatrix}$$

Alternatively, the elements marked in boldface in the matrix A here:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

From the elements selected, a_1, a_1, a_5 and a_6, the max pooling operation selects the maximum value. The result is indicated with B_1

$$B_1 = \max_{i=1,2,5,6} a_i$$

We then need to shift our 2 × 2 window two columns to the right, typically the same number of columns the selected region has, and select the elements marked in bold:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Or, in other words, the smaller matrix

$$\begin{pmatrix} a_3 & a_4 \\ a_7 & a_8 \end{pmatrix}$$

The max-pooling algorithm will then select the maximum of the values and give a result that we will indicate with B_2.

$$B_2 = \max_{i=3,4,7,8} a_i$$

At this point we cannot shift the 2 × 2 region to the right anymore, so we shift it two rows down and start the process again from the left side of A, selecting the elements marked in bold and getting the maximum and calling it B_3.

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

The stride s in this context has the same meaning we have already discussed in convolution. It's simply the number of rows or columns you move your region when selecting the elements. Finally, we select the last region 2 × 2 in the bottom-lower part of A, selecting the elements a_{11}, a_{12}, a_{15}, and a_{16}. We then get the maximum and call it B_4. With the values we obtain in this process, in the example the four values B_1, B_2, B_3 and B_4, we will build an output tensor:

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

In the example, we have $s = 2$. Basically, this operation takes as input a matrix A, a stride s, and a kernel size n_K (the dimension of the region we selected in the example before) and returns a new matrix B with dimensions given by the same formula we discussed for convolution:

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor$$

To reiterate this idea, start from the top-left of matrix A, take a region of dimensions $n_K \times n_K$, apply the max function to the selected elements, then shift the region of s elements toward the right, select a new region again of dimensions $n_K \times n_K$, apply the function to its values, and so on. In Figure 3-11 you can see how you would select the elements from matrix A with stride $s = 2$.

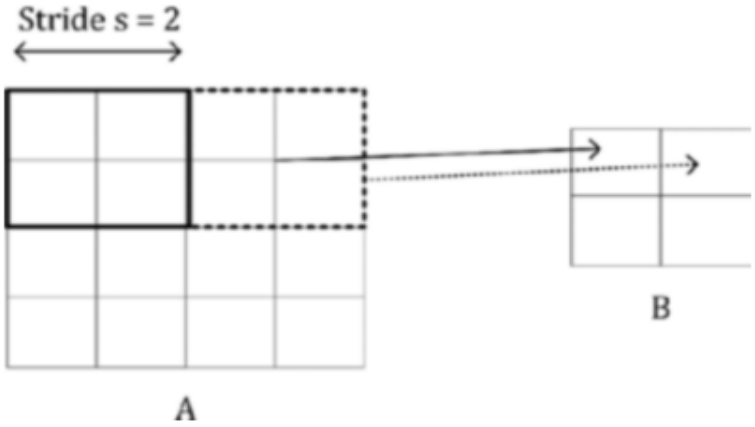

FIGURE 2.10: A visualization of pooling with stride $s = 2$.

Example:

For example, applying max-pooling to the input A:

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

Will get you this result (it's very easy to verify it):

$$B = \begin{pmatrix} 4 & 11 \\ 15 & 21 \end{pmatrix}$$

Since four is the maximum of the values marked in bold.

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

Eleven is the maximum of the values marked in bold here:

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

And so on. It's worth mentioning another way of doing pooling, although it's not as widely used as max-pooling: ***average pooling***. Instead of returning the maximum of the selected values, it returns the average.

Note: The most commonly used pooling operation is max pooling.Average pooling is not as widely used but can be found in specific network architectures.

Padding

Something that's worth mentioning here is *padding*. Sometimes, when dealing with images, it is not optimal to get a result from a convolution operation that has dimensions that are different from the original image. This is when padding is necessary. The idea is straightforward: you add rows of pixels on the top and bottom and columns of pixels on the right and left of the final images so the resulting matrices are the same size as the original. Some strategies fill the added pixels with zeros, with the values of the closest pixels and so on.
For example, in our example, our exout matrix with zero padding would like like this:

```
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Only as a reference, in case you use padding p (the width of the rows and columns you use as padding), the final dimensions of the matrix B, in case of both convolution and pooling, is given by

$$n_B = \left\lfloor \frac{n_A + 2p - n_K}{s} + 1 \right\rfloor$$

Note: When dealing with real images, you always have color images, coded in three channels: *RGB*. That means that convolution and pooling must be done in three dimensions: width, height, and color channel. This will add a layer of complexity to the algorithms.

2.3 Building Blocks of a CNN

Convolution and pooling operations are used to build the layers used in CNNs. In CNNs typically you can find the following layers.

- Convolutional layers
- Pooling layers
- Fully connected layers

2.3.1 Fully connected layers

Fully connected layers is simply a layer where neurons are connected to all neurons of previous and subsequent layers.

2.3.2 Convolutional Layers

A convolutional layer takes as input a tensor (which can be threedimensional, due to the three color channels), for example an image, applies a certain number of

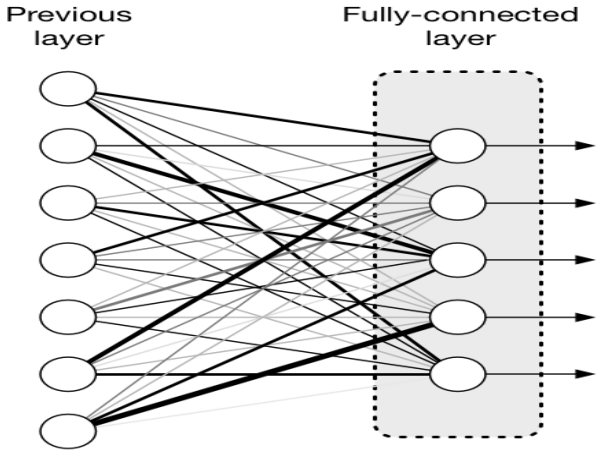

FIGURE 2.11: An example of fully connected layer.

kernels, typically 10, 16, or even more, adds a bias, applies ***ReLu*** activation functions (for example) to introduce nonlinearity to the result of the convolution, and produces an output matrix B.

Now in the previous sections, I showed you some examples of applying convolutions with just one kernel. How can you apply several kernels at the same time? Well, the answer is straightforward. The final tensor (I use now the word tensor since it will not be a simple matrix anymore) B will have not two dimensions but three. Let's indicate the number of kernels you want to apply with n_c (the c is used since sometimes people talk about channels). You simply apply each filter to the input independently and stack the results. So instead of a single matrix B with dimensions $n_B \times n_B$ you get a final tensor $\tilde{B}$ of dimensions $n_B \times n_B \times n_c$. That means that this

$$\tilde{B}_{i,j,1} \quad \forall i, j \in [1, n_B]$$

Will be the output of convolution of the input image with the first kernel, and

$$\tilde{B}_{i,j,2} \quad \forall i, j \in [1, n_B]$$

Will be the output of convolution with the second kernel, and so on. The convolution layer simply transforms the input into an output tensor. However, what are the weights in this layer? The weights, or the parameters that the network learns during the training phase, are the elements of the kernel themselves. We discussed that we have n_c kernels, each of $n_k \times n_k$ dimensions. That means that we have $n_K^2 n_c$ parameter in a convolutional layer.

Note: The number of parameters that you have in a convolutional layer, $n_K^2 n_c$, is independent from the input image size. This fact helps in reducing overfitting, especially when dealing with large input images.

Sometimes this layer is indicated with the word *POOL* and then a number. In our case, we could indicate this layer with *POOL*1. In Figure 2.12, you can see a representation of a convolutional layer. The input image is transformed by applying convolution with n_c kernels in a tensor of dimensions $n_A \times n_A \times n_c$.
Of course, a convolutional layer must not necessarily be placed immediately after

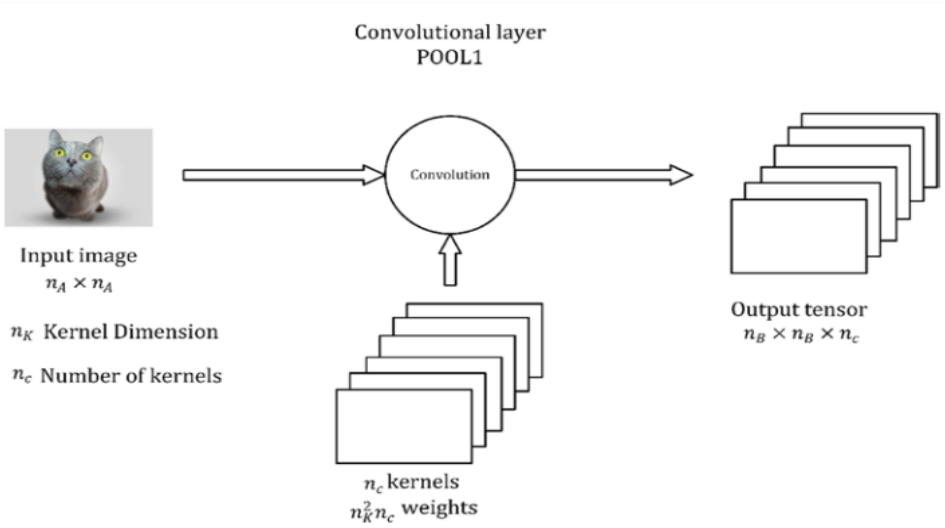

FIGURE 2.12: A representation of a convolutional layer.

the inputs. A convolutional layer may get as input the output of any other layer of course. Keep in mind that usually, the input image will have dimensions $n_A \times n_A \times 3$, since an image in color has three channels: Red, Green, and Blue 2.13. A complete analysis of the tensors involved in a CNN when considering color images is beyond the scope of this book. Very often in diagrams, the layer is simply indicated as a cube or a square.

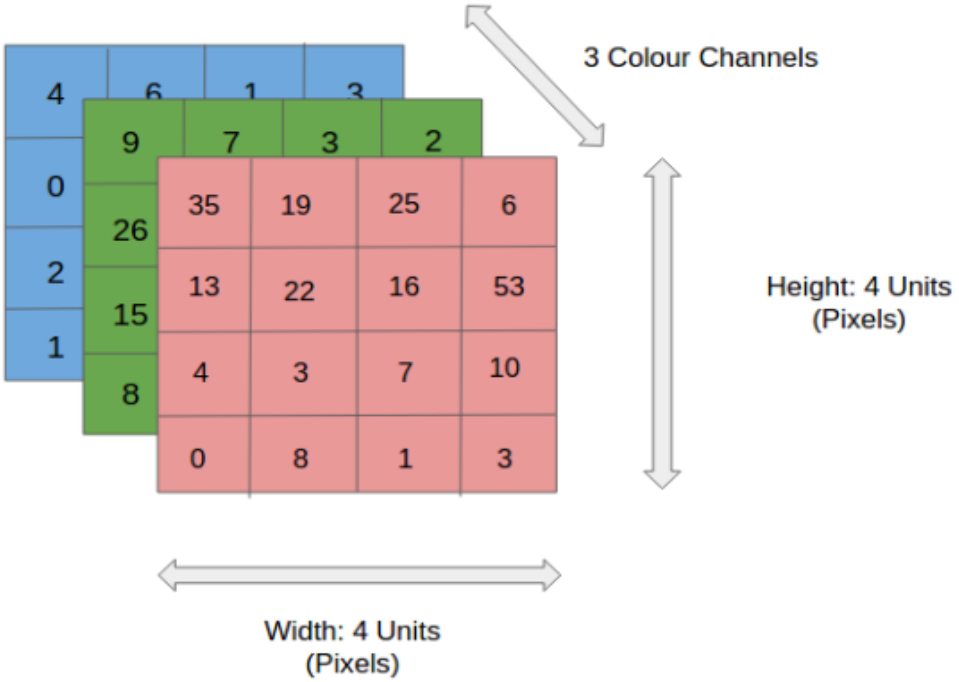

FIGURE 2.13: A $4 \times 4 \times 3$ RGB Image.

2.3.3 Pooling Layers

A pooling layer is usually indicated with *POOL* and a number: for example, *POOL*1. It takes as input a tensor and gives as output another tensor after applying pooling

to the input.

Note A pooling layer has no parameter to learn, but it introduces additional hyperparameters: n_K and stride v. Typically, in pooling layers, you don't use any padding, since one of the reasons to use pooling is often to reduce the dimensionality of the tensors.

2.3.4 Stacking Layers Together

In CNNs you usually stack convolutional and pooling layer together. One after the other. In Figure 2.14, you can see a convolutional and a pooling layer stack. A convolutional *layer* is always followed by a pooling layer.
Sometimes the two together are called a layer. The reason is that a pooling layer has no learnable weights and therefore it is merely seen as a simple operation that is associated with the convolutional layer. So be aware when you read papers or blogs and check what they intend.

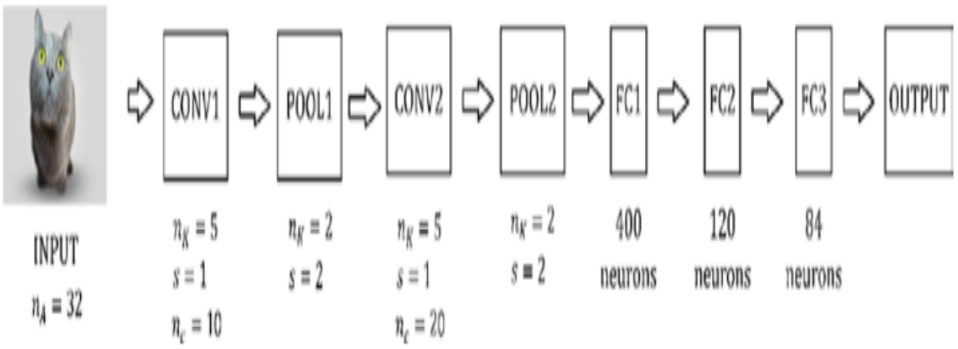

FIGURE 2.14: A representation of how to stack convolutional and pooling layers.

To conclude this part of CNN in Figure 3-14, you can see an example of a CNN. In Figure 3-14, you see an example like the very famous LeNet-5 network.
You have the inputs, then two times convolution-pooling layer, then three fully connected layers, and then an output layers, with a $softmax$ activation function to perform multiclass classification. I put some indicative numbers in the figure to give you an idea of the size of the different layers.

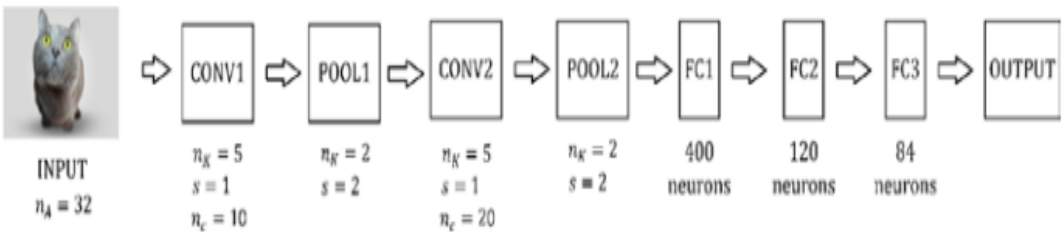

FIGURE 2.15: A representation of a CNN similar to the famous LeNet-5 network.

2.4 Number of Weights in a CNN

It is important to point out where the weights in a CNN are in the different layers.

2.4.1 Convolutional Layer

In a convolutional layer, the parameters that are learned are the filters themselves. For example, if you have 32 filters, each of dimension 5x5, you will get 32x5x5=832 learnable parameters, since for each filter there is also a bias term that you will need to add. Note that this number is not dependent on the input image size. In a typical feed-forward neural network, the number of weights in the first layer is dependent on the input size, but not here.

The number of weights in a convolutional layer is, in general terms, given by the following:

$$n_C \cdot n_K \cdot n_K + n_C$$

2.4.2 Pooling Layer

The pooling layer has no learnable parameters, and as mentioned, this is the reason it's typically associated with the convolutional layer. In this layer (operation), there are no learnable weights.

2.4.3 Dense Layer

In this layer, the weights are the ones you know from traditional feedforward networks. So the number depends on the number of neurons and the number of neurons in the preceding and subsequent layers.

Notes: The only layers in a CNN that have learnable parameters are the convolutional and dense layers.

Chapter 3

Image Classification : An Implementation in Python

In this chapter we will be using a dataset named **CIFAR-10** dataset. The **CIFAR-10** dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research.
Brief information about CIFAR-10 dataset:

- The CIFAR-10 dataset contains $60,000$ 32×32 color images in 10 different classes.
- The 10 different classes represent *airplanes*, *cars*, *birds*, *cats*, *deer*, *dogs*, *frogs*, *horses*, *ships*, and *trucks*
- There are $6,000$ images of each class.

In the code below, we have used **Keras** library to build an image cassification model trained on the CIFAR-10 dataset. It uses the following layers/functions:

- **For building the Model** - CNN, Maxpooling and Dense Layers.
- **For Activation Function** - ReLU (in CNN layers for handling image pixels) and Softmax (for final classification).
- **For handling Overfitting (Regularizing)** - DropOut Layer.
- **For normalizing/standardizing** the inputs between the layers (within the network) and hence accelerating the training, providing regularization and reducing the generalization error - Batch Normalization Layer.

3.1 Python Implementation

3.1.1 Importing Dependencies

```
import keras
from keras.datasets import cifar10
from keras.models import Sequential
from keras import datasets, layers, models
from keras.utils import np_utils
from keras import regularizers
from keras.layers import Dense, Dropout, BatchNormalization
import matplotlib.pyplot as plt
import numpy as np
```

3.1.2 Download and prepare the CIFAR10 dataset

```
(train_images, train_labels), (test_images, test_labels) = datasets.
    cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 3s 0us/step
```

FIGURE 3.1: Jupyter Notebook

3.1.3 EDA (Exploratory Data Analysis)

```
# Checking the number of rows (records) and columns (features)
print(train_images.shape)
print(train_labels.shape)
print(test_images.shape)
print(test_labels.shape)
```

```
(50000, 32, 32, 3)
(50000, 1)
(10000, 32, 32, 3)
(10000, 1)
```

FIGURE 3.2: Jupyter Notebook

```
# Creating a list of all the class labels
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

```
# Visualizing some of the images from the training dataset
plt.figure(figsize=[10,10])
for i in range (25):    # for first 25 images
  plt.subplot(5, 5, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.grid(False)
  plt.imshow(train_images[i], cmap=plt.cm.binary)
  plt.xlabel(class_names[train_labels[i][0]])

plt.show()
```

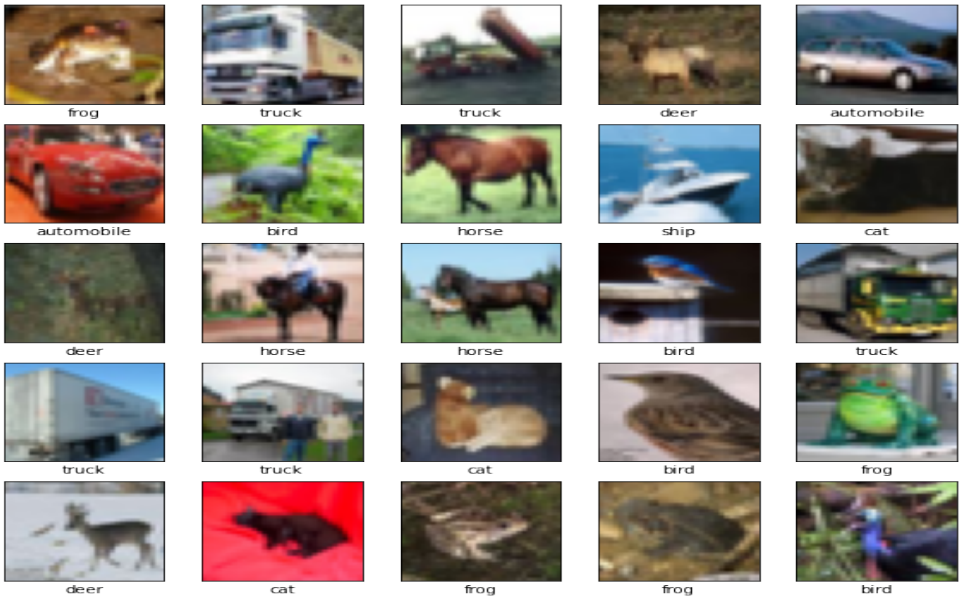

FIGURE 3.3: Jupyter Notebook

3.1.4 Data Preprocessing

- The reason for Standardizing/Normalizing is to convert all pixel values to values between 0 and 1.
- The reason for converting type to float is that to-categorical (one hot encoding) needs the data to be of type float by default.
- The reason for using to-categorical is that the loss function that we will be using in this code (categorical-crossentropy) when compiling the model needs data to be one hot encoded.

```
# Converting the pixels data to float type
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

# Standardizing (255 is the total number of pixels an image can have)
train_images = train_images / 255
test_images = test_images / 255

# One hot encoding the target class (labels)
num_classes = 10
train_labels = np_utils.to_categorical(train_labels, num_classes)
test_labels = np_utils.to_categorical(test_labels, num_classes)
```

3.1.5 Building the CNN Model using Keras

Setting up the Layers

```

# Creating a sequential model and adding layers to it

model = Sequential()

```

```
model.add(layers.Conv2D(32, (3,3), padding='same', activation='relu',
    input_shape=(32,32,3)))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(32, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(64, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(64, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.5))

model.add(layers.Conv2D(128, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(128, (3,3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(layers.Dropout(0.5))

model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(num_classes, activation='softmax'))  #
    num_classes = 10

# Checking the model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
```

FIGURE 3.4: Jupyter Notebook

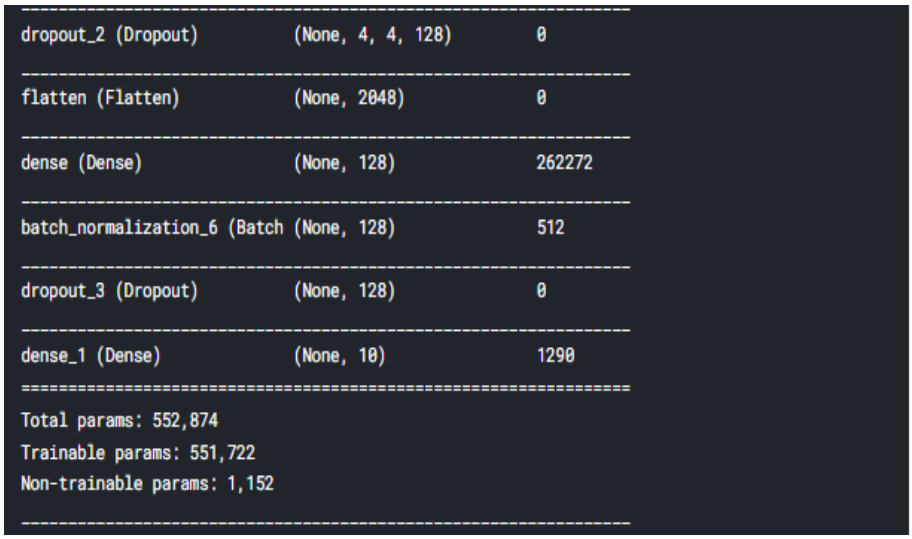

FIGURE 3.5: Jupyter Notebook

Compiling the Model

- Optimizer used during Back Propagation for weight and bias adjustment - Adam (adjusts the learning rate adaptively).
- Loss Function used - Categorical Crossentropy (used when multiple categories/-classes are present).
- Metrics used for evaluation - Accuracy.

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

Fitting the Model

- Batch Size is used for Adam optimizer.
- Epochs - One epoch is one complete cycle (forward pass + backward pass).

```
history = model.fit(train_images, train_labels, batch_size=64, epochs
    =100,
                    validation_data=(test_images, test_labels))
```

```
Epoch 92/100
782/782 [==============================] - 7s 8ms/step - loss: 0.2610 - accuracy: 0.9094 - v
al_loss: 0.4638 - val_accuracy: 0.8585
Epoch 93/100
782/782 [==============================] - 7s 9ms/step - loss: 0.2536 - accuracy: 0.9106 - v
al_loss: 0.4060 - val_accuracy: 0.8747
Epoch 94/100
782/782 [==============================] - 7s 9ms/step - loss: 0.2570 - accuracy: 0.9099 - v
al_loss: 0.4091 - val_accuracy: 0.8740
Epoch 95/100
211/782 [=======>......................] - ETA: 4s - loss: 0.2470 - accuracy: 0.9143
```

FIGURE 3.6: Jupyter Notebook

3.1.6 Visualizing the Evaluation

- Loss Curve - Comparing the Training Loss with the Testing Loss over increasing Epochs.
- Accuracy Curve - Comparing the Training Accuracy with the Testing Accuracy over increasing Epochs.

```
# Loss curve
plt.figure(figsize=[6,4])
plt.plot(history.history['loss'], 'black', linewidth=2.0)
plt.plot(history.history['val_loss'], 'green', linewidth=2.0)
plt.legend(['Training Loss', 'Validation Loss'], fontsize=14)
plt.xlabel('Epochs', fontsize=10)
plt.ylabel('Loss', fontsize=10)
plt.title('Loss Curves', fontsize=12)
```

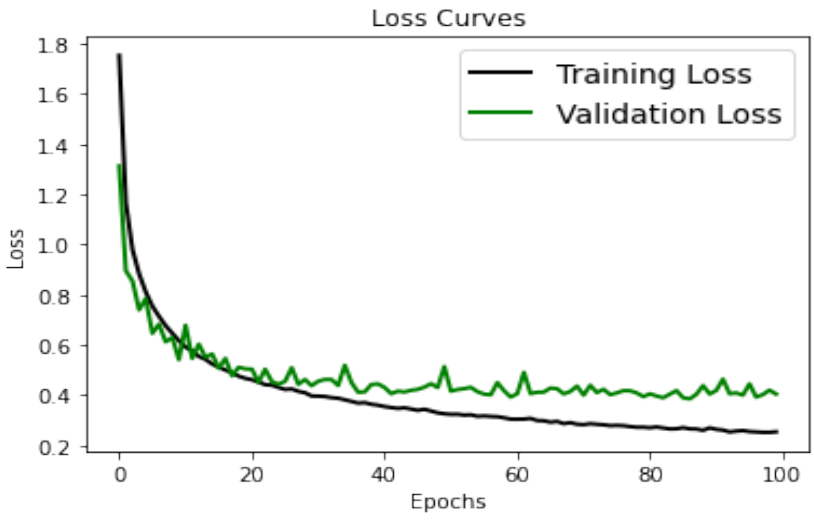

FIGURE 3.7: Jupyter Notebook

```
# Accuracy curve
plt.figure(figsize=[6,4])
plt.plot(history.history['accuracy'], 'black', linewidth=2.0)
plt.plot(history.history['val_accuracy'], 'blue', linewidth=2.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'], fontsize=14)
plt.xlabel('Epochs', fontsize=10)
```

```
plt.ylabel('Accuracy', fontsize=10)
plt.title('Accuracy Curves', fontsize=12)
```

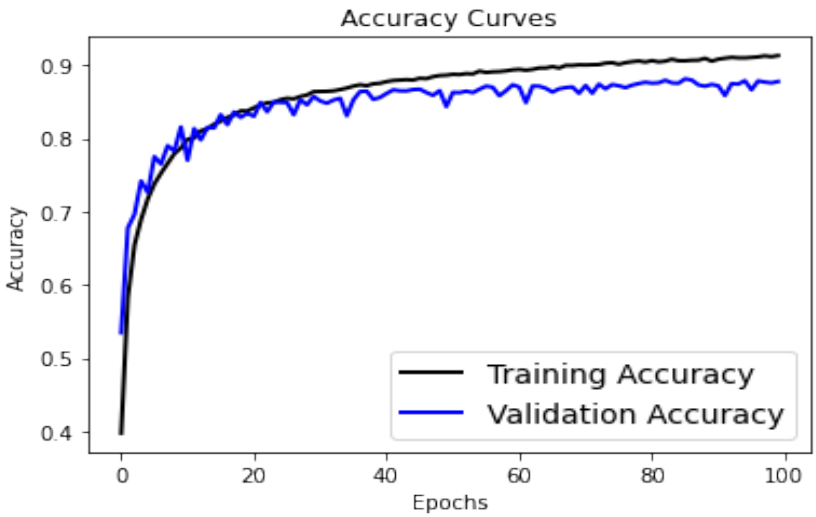

FIGURE 3.8: Jupyter Notebook

3.1.7 Predicting the Result

Let's take 25 images from the testing data and see how many of it we predicted correctly.

```
# Making the Predictions
pred = model.predict(test_images)
print(pred)

# Converting the predictions into label index
pred_classes = np.argmax(pred, axis=1)
print(pred_classes)
```

```
[[2.3109624e-06 5.0927770e-06 1.5846036e-04 ... 1.1763057e-06
  2.4639112e-06 5.9165342e-07]
 [4.8080442e-06 5.4957319e-02 1.5304867e-09 ... 4.3609136e-10
  9.4503629e-01 1.5745940e-06]
 [3.9746395e-05 1.1915377e-03 4.5444347e-07 ... 5.2812752e-08
  9.9876142e-01 3.3142740e-06]
 ...
 [3.9121325e-08 1.2354816e-07 4.7954560e-05 ... 2.0266171e-04
  8.0094800e-08 1.6819777e-06]
 [1.4044832e-03 9.9727124e-01 1.6699676e-05 ... 1.2907191e-05
  8.8348979e-06 2.6951061e-04]
 [4.3556261e-10 3.1211504e-09 4.6839427e-10 ... 9.9999976e-01
  1.5790504e-10 5.9609200e-11]]
[3 8 8 ... 5 1 7]
```

FIGURE 3.9: Jupyter Notebook

```
# Plotting the Actual vs. Predicted results

fig, axes = plt.subplots(5, 5, figsize=(15,15))
axes = axes.ravel()
```

```
5
6  for i in np.arange(0, 25):
7      axes[i].imshow(test_images[i])
8      axes[i].set_title("True: %s \nPredict: %s" % (class_names[np.argmax
       (test_labels[i])], class_names[pred_classes[i]]))
9      axes[i].axis('off')
10     plt.subplots_adjust(wspace=1)
```

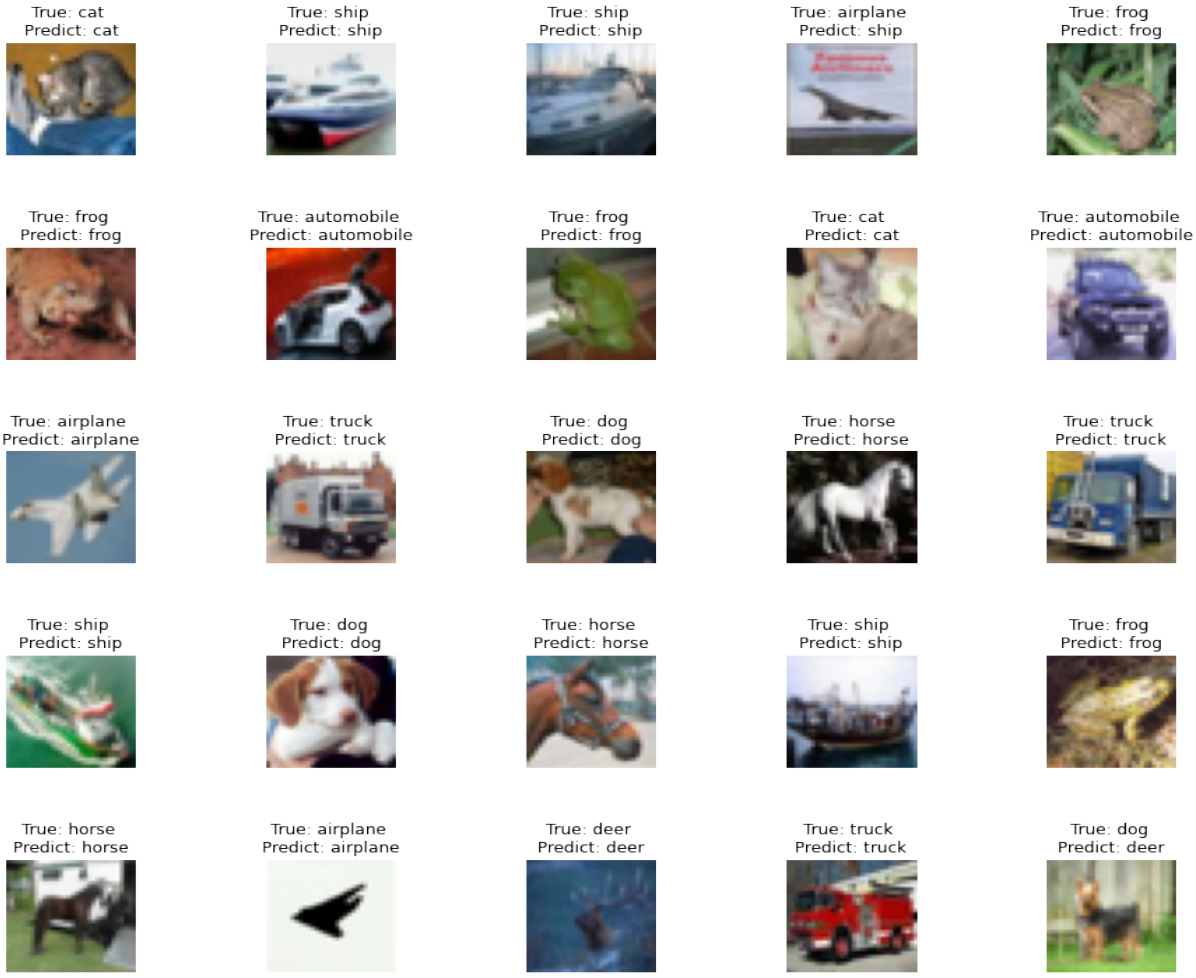

FIGURE 3.10: Jupyter Notebook

3.2 Conclusion

The results obtained in this implementation are important as they indicate that it is possible to increase the accuracy of CNN models by easily running and modifying its traditional structure with the use of programming language. One of the interesting conclusions is the ratio between accuracy and running time, as the CNN structure obtained the best running time to accuracy ratio. Basing on the possessed components, the AI implementers would have to decide if it is worth to go for the more robust models.

Bibliography

[1] Umberto Michelucci, *"Advanced Applied Deep Learning - Convolutional Neural Networks and Object Detection"*, Umberto Michelucci TOELT LLC, Dübendorf, Switzerland.

[2] Umberto Michelucci. *"Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks"*, Umberto Michelucci toelt.ai, Dübendorf, Switzerland. URL: https://link.springer.com/book/10.1007%2F978-1-4842-3790-8

[3] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. *"A Guide to Convolutional Neural Networks for Computer Vision"*.URL: https://www.morganclaypool.com

[4] Rowel Atienza. *"Advanced Deep Learning with Keras"* (2018). URL: https://www.packtpub.com/product/advanced-deep-learning-with-keras/9781788629416

[5] Ajala Sunday Adeyinka. *"Image Classification Using Convolutional Neural Networks"*, Norfolk State University, 700 Park Avenue, Norfolk, USA, 23504. URL: https://www.researchgate.net/profile/Sunday-Ajala/research