المدرسة الوطنية للعلوم التطبيقية
ⵜⵉⵍⵍⵉ ⵜⴰⵙⴷⴰⵡⵉⵜ ⵜⵓⵙⵙⵓⵔⵉⵏ ⵜⵓⵏⵉⵡⵓⵏ
Ecole Nationale des Sciences Appliquées de Fès

جامعة سيدي محمد بن عبد الله
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵙⵉⴷⵉ ⵎⵓⵃⴰⵎⴻⴷ ⴱⵏ ⵄⴱⴷⵍⵍⴰⵀ
Université Sidi Mohamed Ben Abdellah

Final Project Report

Artificial Intelligent

Image Classification Using Convolutional Neural Network CNN

Embedded Systems and Industrial Computing Engineering

STUDENTS:

BOUGRINE Imane

EL KANTRI Youssef

MANSOURI Anas

RIZKI Younes

SUPERVISOR:

CHOUGRAD Hiba

2019-2020

ACKNOWLEDGEMENTS

We would like to acknowledge our supervisor Prof. CHOUGRAD Hiba for providing us the necessary guidance and valuable information to complete this research project. We value her efforts in class to teach us the fundamentals of machine learning. Learning from her knowledge helped us to become passionate about our research topic.

We would like also to acknowledge the support provided by our family during the preparation of this project.

ABSTRACT

The use of deep learning and specifically neural networks is a growing trend in software development, and has grown immensely in the last couple of years in the light of an increasing need to handle big data and large information flows. Deep learning has a broad area of application, such as human-computer interaction, predicting stock prices, real-time translation, self-driving vehicles and image classification. Large companies such as Microsoft and Google have already implemented machine learning in some of their commercial products such as their search engines, and their intelligent personal assistants Cortana and Google Assistant. The main purpose of the work presented in this paper, is to apply the concept of a Deep Learning algorithm namely, Convolutional neural networks (CNN) in image classification. The algorithm is tested on a standard dataset named CIFAR-10 dataset. The performance of the algorithm is evaluated based on the quality metric known as classification accuracy and confusion matrix. The experimental result analysis based on the quality metrics proves that the algorithm (CNN) gives fairly good classification accuracy for all the tested datasets.

RESUME

L'utilisation de l'apprentissage profond (Deep learning) et en particulier les réseaux de neurones est une tendance croissante dans le développement de logiciels, et a énormément évolué au cours des deux dernières années à cause d'un besoin croissant de gérer de grandes quantités de données et les grands flux d'informations. Le Deep Learning a un domaine d'application large, comme l'interaction homme-ordinateur, la prévision des cours des actions, la traduction en temps réel, les véhicules autonomes et la classification d'images. De grandes entreprises telles que Microsoft et Google ont déjà mis en œuvre l'apprentissage automatique dans certains de leurs produits commerciaux tels que leurs moteurs de recherche, et leurs assistants personnels intelligents Cortana et Google Assistant. L'objectif principal du travail présenté dans ce rapport est d'appliquer le concept d'un algorithme d'apprentissage profond, à savoir les réseaux de neurones convolutifs (CNN) dans la classification d'images. L'algorithme est testé sur un ensemble de données standard nommé ensemble de données CIFAR-10. Les performances de l'algorithme sont évaluées en se basant sur les métriques de qualité appelées Accuracy et matrice de confusion. L'analyse des résultats expérimentaux basée sur les métriques de qualité prouve que l'algorithme (CNN) donne une assez bonne précision de classification pour tous les ensembles de données testés.

CONTENTS

LIST OF FIGURES

LIST OF ABREVIATION

ANN: Artificial Neural Networks

CNN: Convolutional Neural Networks

CONV: Convolution

ReLU: Rectified Linear Unit

I. INTRODUCTION

Image classification is one of the most fundamental problems in Machine Learning. It is the core foundation for bigger problems such as Computer Vision, Face Recognition System, or Self-driving car. With the development of Convolutional Neural Network (CNN), researchers have achieved good performance on the image recognition task.

The purpose of this paper is to look at CNN architecture, and test its performance on the CIFAR-10 Dataset.

II. BACKGROUND

This chapter will introduce the rationale behind this study and its scope, as well as providing the necessary theoretical background for someone completely new to machine learning to understand the project's implementation. The deep learning frameworks used in this study will be presented and introduced as well.

So, in order to explain why convolutional neural networks "CNN" are chosen to solve the problem of image classification, we will start with some basic concepts that shows how images were classified in the computer vision before the arrival of CNN model.
The first concept is the **Artificial Neural Network:** What is ANN?

A. Artificial Neural Network

Artificial neural networks (ANNs) or connectionist systems are computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.

ANN is a set of connected organized in layers:

- **Input layer:** brings the initial data into the system for further processing by subsequent layers of artificial neurons.
- **Hidden layer**: is a layer between input layers and output layers, where artificial neurons take a set of weighted inputs and produce an output through an activation function
- **Output layer**: the last layer of neurons that produces given outputs for the program.

ANN may contain 1 or more hidden layers.

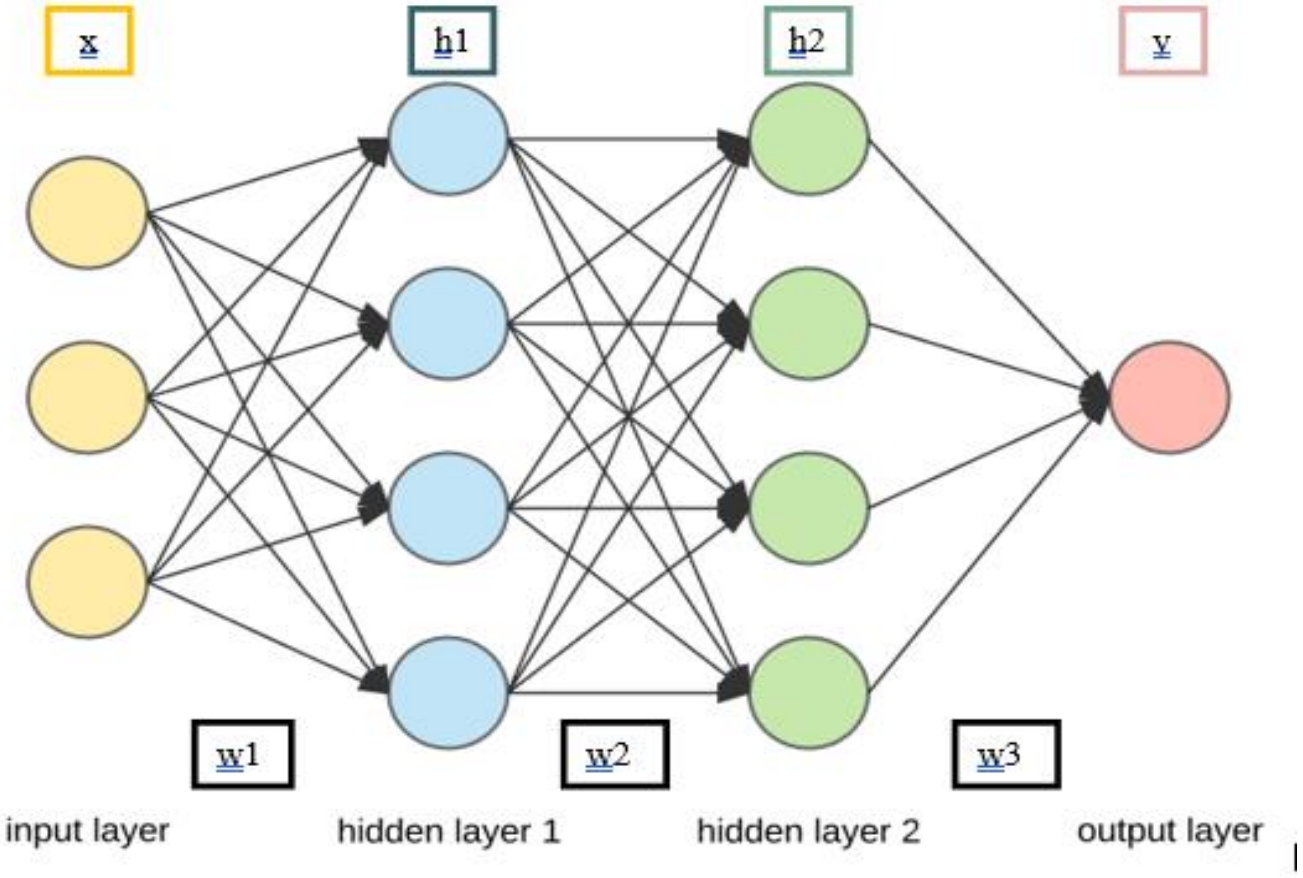


Figure 1: artificial neural network with two hidden layers

Given an input vector x, we compute a dot-product with the first weight matrix **W1** and apply the activation function to the result of this dot-product. The result is a new vector **h1**, which represents the values of the neurons in the first layer. This vector **h1** is used as new input vector for the next layer, where the same operations are performed again. This is repeated until we get the final output vector y, which is considered as the prediction of the neural network.

The entire set of operations can be represented by the following equations,
where σ represent an arbitrary activation function:

$$h1 = \sigma(x, W1)$$

$$h2 = \sigma(h1, W2)$$

$$y = \sigma(h2, W3)$$

As mentioned above , neural networks use an activation function , what is an activation function and why do we need it, in other word why not just use the output as it is without need to this function ?

The answer is that The purpose of an activation function is to add some kind of non-linear property to the function, which is a neural network. Without the activation functions, the neural network could perform only linear mappings from inputs *x* to the outputs *y*. Why is this so?

Without the activation functions, the only mathematical operation during the forward propagation would be dot-products between an input vector and a weight matrix. Since a single dot product is a linear operation, successive dot products would be nothing more than multiple linear operations repeated one after the other. And successive linear operations can be considered as a one single learn operation.

In order to be able to compute really interesting stuff, neural networks must be able to approximate nonlinear relations from input features to output labels. Usually, the more complex the data is we are trying to learn something from, the more non-linear the mapping of features to the ground truth label is.

Most common activation functions are:

- ✓ **Sigmoid**:

The sigmoid function maps the incoming inputs to a range between 0 and 1.

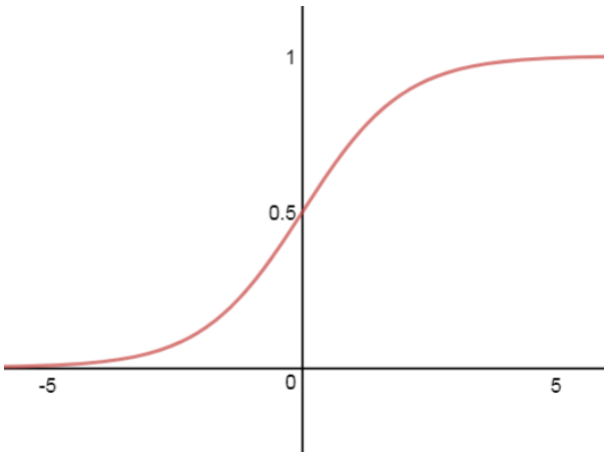


Figure 2: sigmoid function

This function is defined as followed:

$$sigmoid(x) = \frac{e^x}{1 + e^x}$$

- ✓ **ReLU**

The Rectified Linear Unit or just simply ReLU has become very popular in the last few years. The activation is simply thresholded at zero: *R(x) = max (0, x)* that means

$$if\ x < 0,\ R(x) = 0\ and\ if\ x >= 0,\ R(x) = x$$

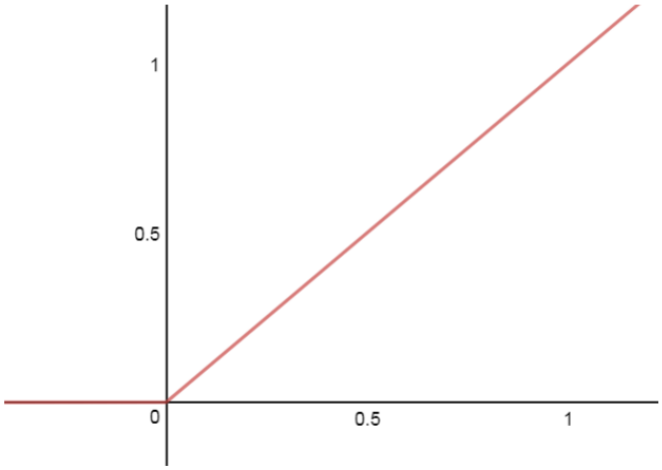


Figure 3: Relu function

There more than those functions and their use depends on the task we are dealing with.

The second concept is **how does the computer see an image**?

- ✓ **Softmax**

Softmax function calculates the probabilities distribution of the event over 'n' different events.

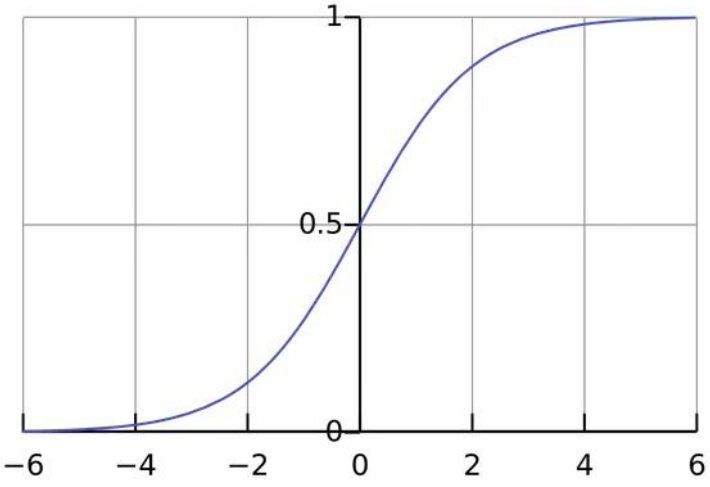


Figure 4: Softmax function

B. Representation of an image

Image classification is the task of taking an input image and outputting a class (a cat, dog, etc) or a probability of classes that best describes the image. We humans learn how to do this task within the first month of us being born, and for the rest of our lives it comes naturally and effortlessly to us. We're able to quickly identify the environment we are in as well as the objects that surround us, all without even consciously noticing. These skills of being able to quickly recognize patterns, generalize from prior knowledge, and adapt to different image environments are ones that we do not share with our fellow machines, because they see the images quiet differently:

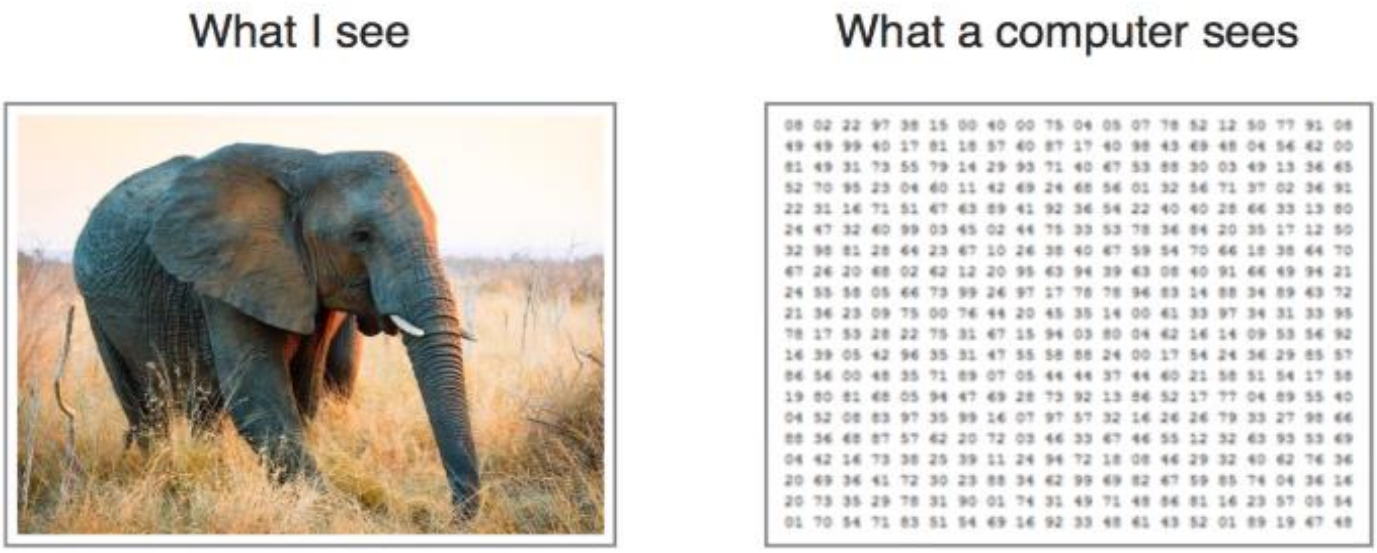


Figure 5: How an image is seen by human being and by the computer

Instead of the image, the computer sees a matrix of pixels. For example, if image size is 21x21 and if it's a colored image (not a grayscale image), then in this case, the real size of the image is 21x21x3. Where 21 is width, next 21 is height and 3 is RGB channel values. The computer is assigned a value from 0 to 255 to each of these numbers. This value describes the intensity of the pixel at each point.

So, to classify this image the matrix will be flattered to be connected to the neural network, that means that the input of our neural network will be the flattered matrix as it's mentioned bellow:

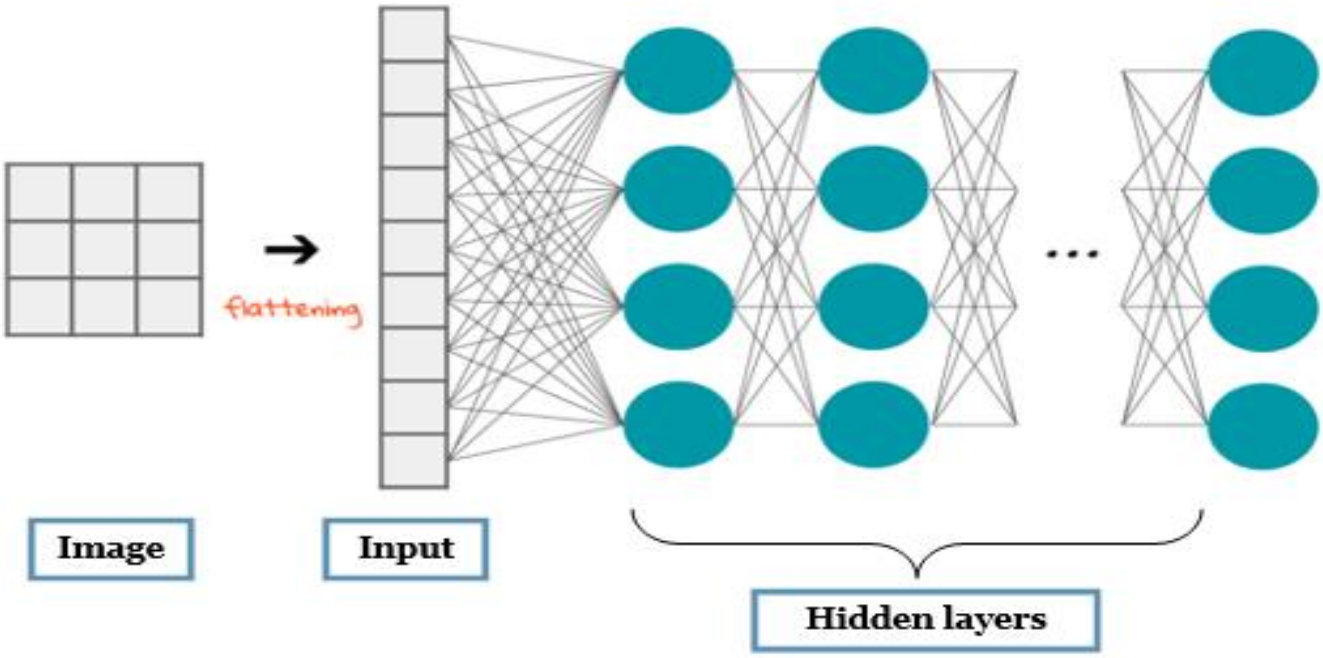


Figure 6: Overview of a classic image classifier

The classic method of classifying images was based on flattering image and threating it by neural Network but this method has a lot of disadvantages.

C. Problem & solution

Artificial neural networks have shown their effectiveness as a learning technique for data classification. They are indeed able to approximate complex non-linear functions to process large data. But they have inconvenient:

- Conventionally, the layers of a neural network are completely connected, that is to say that the value of a neuron of a layer **n** will depend on the values of all the neurons of the layer (n – 1). Thus, the number of connections (and therefore the number of weight and of parameters) can be very large. For example, for an image of size 15 × 15, the dimension of the entry of an ANN is 15*15*3=675. If the hidden layer has 100 neurons, then the number of parameters of this layer is 100 × 675 = 67500. The number of parameters will thus **increase exponentially** with the size of the size of the entrance (pictures). This great complexity of the network requires having many learning samples, which is often not the case. The network will therefore tend to overfit, and will therefore offer a poor generalization capacity
- Another shortcoming of ANN for image applications is that the complexity of the treatment will require too much time and resources (very powerful GPUs)
- Finally, ANN does not consider the correlation between pixels of an image, which is a very important element for pattern recognition.

To deal with these problems, Convolutional Neural Network has been invented. CNN are an extension of ANN to effectively respond to the main problems already mentioned. They are designed to automatically extract the characteristics of the input images, they are invariant to slight image distortions, and they implement the concept of **weight sharing** making it possible to considerably reduce the number of network parameters.

III. CONVOLUTIONAL NEURAL NETWORK

In this chapter we will define the Convolutional Neural Network then we will describe its architecture.

A. Definition of CNN

CNN or Convolutional Neural Network is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis and natural language processing (NLP).

B. CNN Architecture

A CNN architecture is formed by a stack of independent processing layers:

- Convolutional layer (CONV) which processes the data of a receiver field

In general, the CNN architecture is composed of two main parts:

⇨ The first one is the convolutional part; it functions as an extractor of image characteristics. An image is passed through a succession of filters, or convolution kernels, creating new images called convolution maps. Some intermediate filters reduce the resolution of the image by a local maximum operation. In the end, the convolution maps are flattened and concatenated into a vector of characteristics, called the CNN code

⇨ This CNN code at the output of the convolutional part is then connected to the input of a second part, consisting of fully connected layers (multilayer perceptron). The role of this part is to combine the characteristics of the CNN code to classify the image as it's illustrated in the figure bellow:

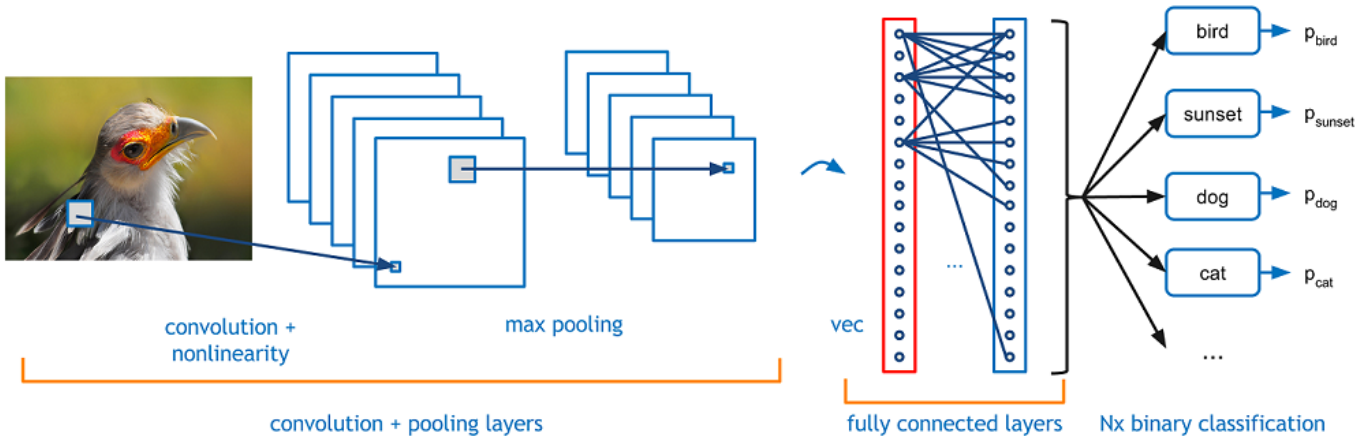


Figure 7: Overview of the CNN architecture

1. Convolution

The convolutional layer is the basic building block of a CNN.

Originally, a convolution is a mathematical tool (we talk about the convolutional product) widely used in image editing because it allows to bring out characteristics:

- ⇨ Highlighting vertical, horizontal, diagonal lines…
- ⇨ Blurry of the image (blur), smoothing of textures
- ⇨ Color inversion
- ⇨ etc…

The convolution is made by applying a **"filter".** In fact, a convolution simply takes as input an image and a filter (which is another image), performs a calculation, then returns a new image (generally smaller) that shows characteristics of the input image.

These characteristics represents often a horizontal or vertical edge of the input image as it's shown by this figure:

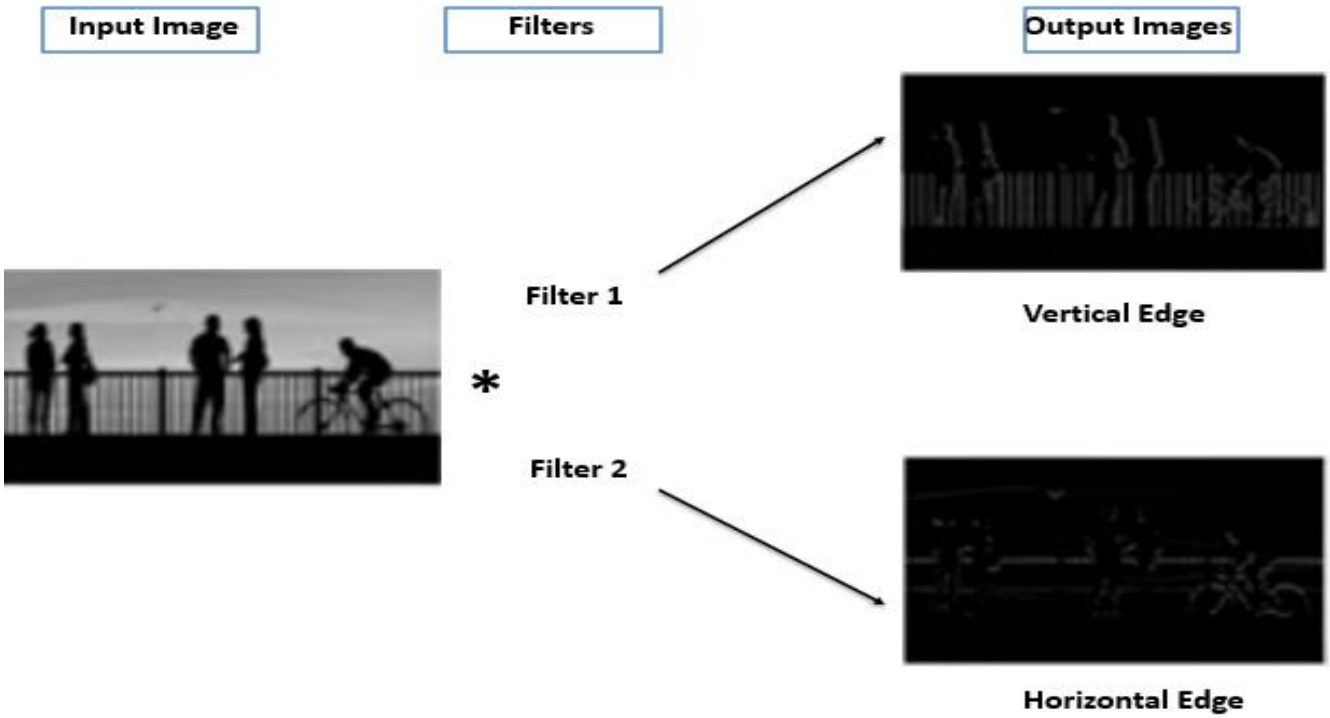


Figure 8: the principle of convolution : edge detection

But how do we detect these edges? To illustrate this, we will take an example:

- **Edge detection example:**

Let's take a 6 X 6 grayscale image (i.e. only one channel):

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

Figure 9 : The input image

Next, we convolve this 6 X 6 matrix with a 3 X 3 filter:

1	0	-1
1	0	-1
1	0	-1

Figure 10 : vertical filter

After the convolution, we will get a 4 X 4 image. The first element of the 4 X 4 matrix will be calculated as:

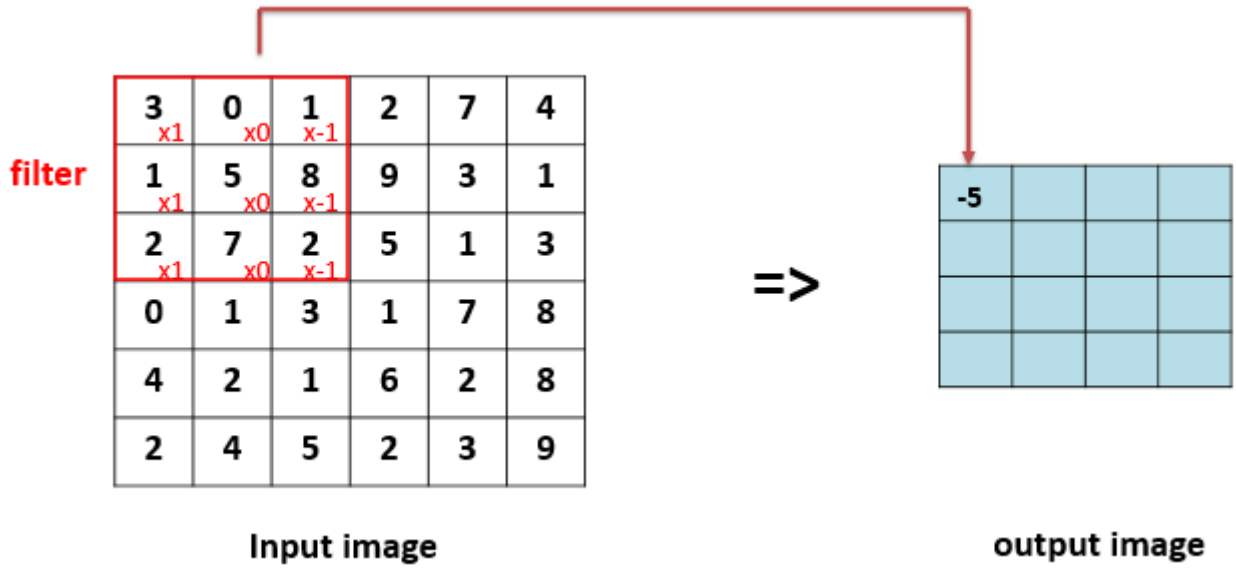


Figure 11: First element in the output matrix

So, we take the first 3 X 3 matrix from the 6 X 6 image and multiply it with the filter. Now, the first element of the 4 X 4 output will be the sum of the element-wise product of these values, i.e. 3*1 + 0 + 1*-1 + 1*1 + 5*0 + 8*-1 + 2*1 + 7*0 + 2*-1 = -5. To calculate the second element of the 4 X 4 output, we will shift our filter one step towards the right and again get the sum of the element-wise product.

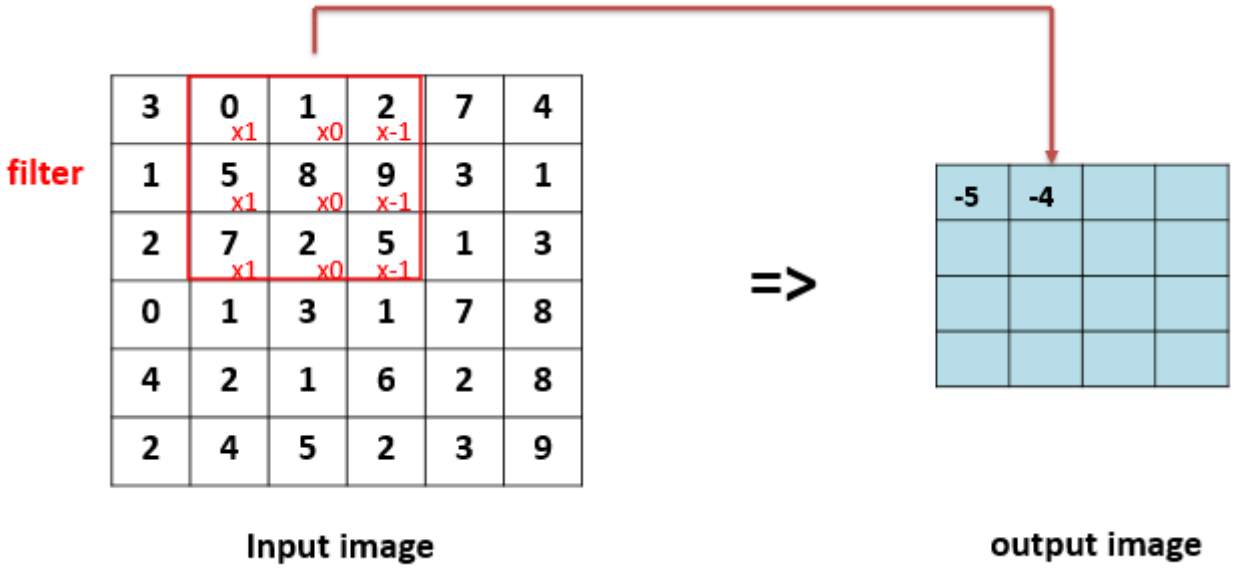


Figure 12: Second element in the output matrix

And suchlike until we get the final image.

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

Figure 13: The output result

- **Filter types**

There are plenty of filter types that can detect different edges such as:

1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

Figure 14: Common filter types

Instead of using these filters, we can create our own as well and treat them as a parameter which the model will learn.

We have seen that convolving an input of **6x6** dimension with a **3x3** filter results in **4x4** output. We can generalize it and say that if the input is **nxn** and the filter size is **fxf**, then the output size will be $(n-f+1)$x$(n-f+1)$**:**

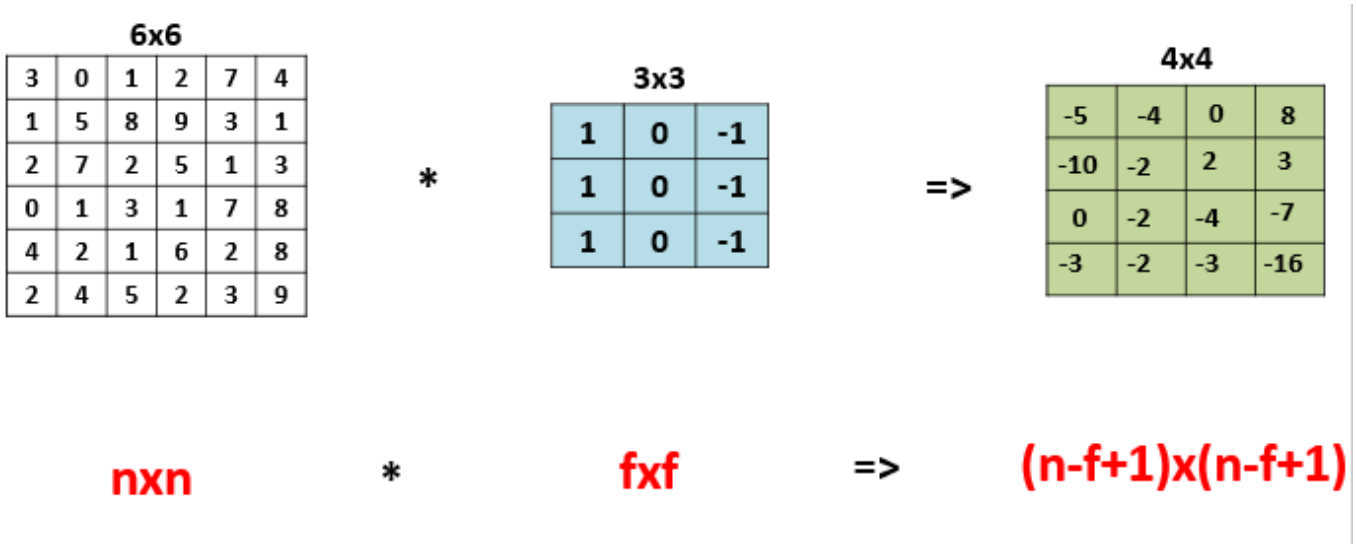


Figure 15: mathematic formula of the convolution

There are primarily two disadvantages here:

1. Every time we apply a convolutional operation, the size of the image shrinks
2. Pixels in the corner of the image are used only a few numbers of times during the convolution as compared to the central pixels. Hence, we do not focus too much on the corners since that can lead to information loss.

- **Padding**

To deal with these issues, we can pad the image with an additional border, i.e., we add one more pixel all around the edges. This means that the input will be an 8x8 matrix (instead of a 6x6 matrix). Applying convolution of 3 X 3 on it will result in a 6 X 6 matrix which is the original shape of the image. Then this gives us the mathematic formula of the size of the output image: $(n + 2p - f + 1)\mathrm{X}(n + 2p - f + 1)$,which p is the padding added.

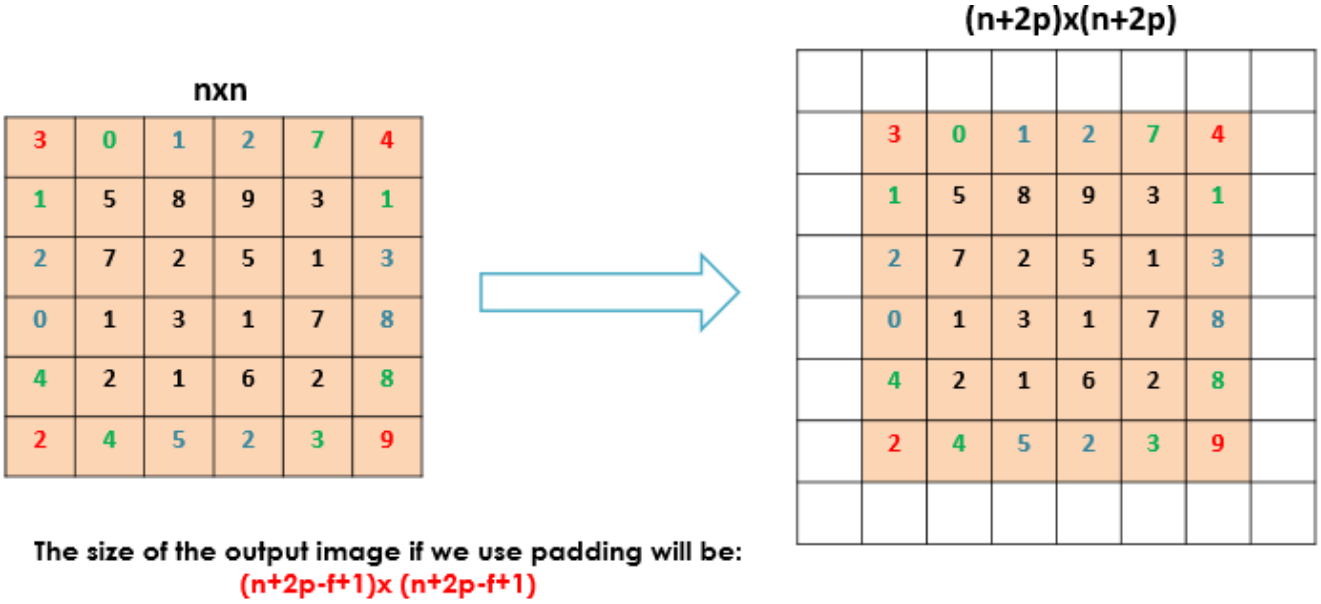


Figure 16: Mathematic formula of the size of the output image using padding

Finally, we can say that there are two main types of padding: **valid** and **same** padding**.**

- **Valid** means no padding. The output size of the convolutional layer shrinks depending on the input size & kernel size.
- **Same** means using padding. The output size of the convolutional layer can have the same size of the input image if we choose $p = \frac{f-1}{2}$

- **Strided convolution**

Until now, we used a stride of 1, that means that the filter is shifted one step towards the right, but we can modify the stride and take s, which means that the output image size will be:

$$\left(\frac{(n+2p-f)}{s}+1\right) \mathrm{X} \left(\frac{(n+2p-f)}{s}+1\right)$$

- **Convolution over volume**

The input image has 3 parameters:

⇨ Number of channels: the number of color spaces in which images exist , it can be Grayscale (1 channel), RGB (3 Channels) , CMYK (4 channels) …etc.
⇨ The height units (or pixels)
⇨ The width units

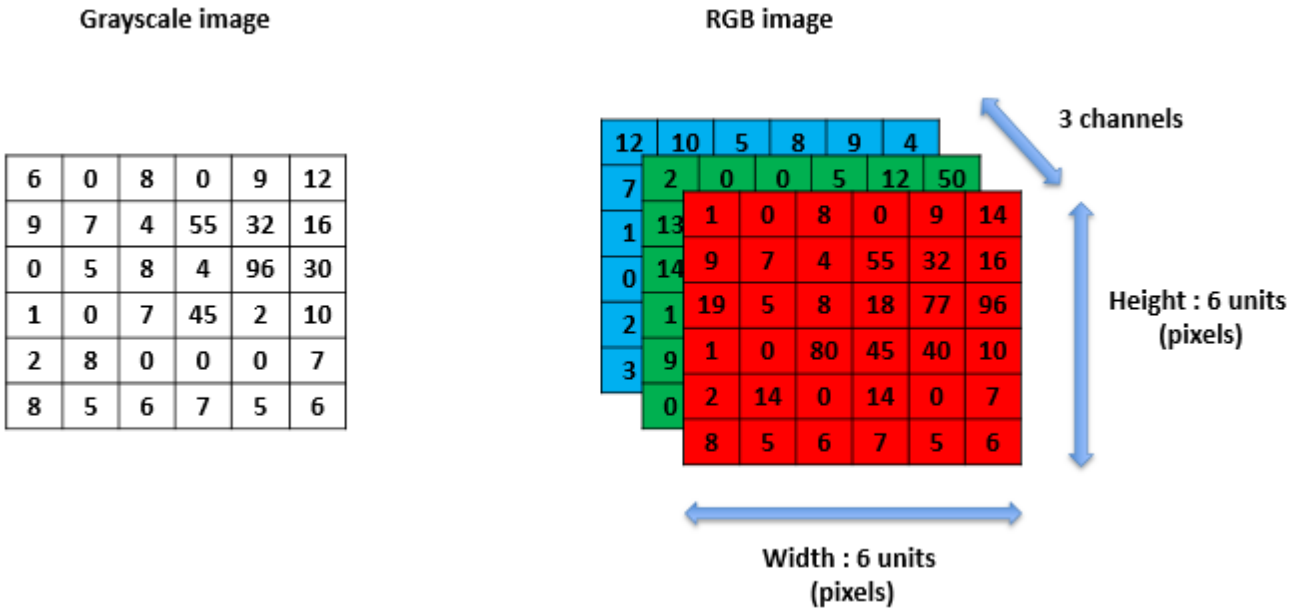


Figure 17: Types of the input images

Previously we considered the grayscale image, in this section we will take an RGB Image (3D image) and then we will convolve it with a 3x3x3 filter instead of 3x3 filter because the number of channels in the input and filter should be same.

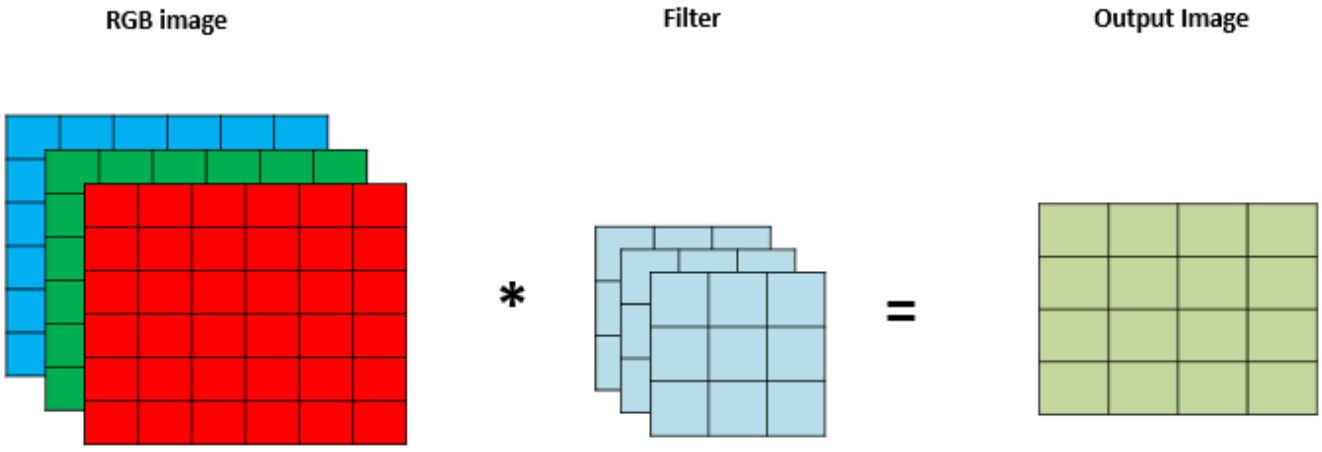


Figure 18 : convolution of an RGB image

Since there are three channels in the input, the filter will consequently also have three channels. After convolution, the output shape is a 4x4 matrix. So, the first element of the output is the **sum** of the element-wise product of the first 27 values from the input (9 values from each channel) and the 27 values from the filter. After that we convolve over the entire image.

Instead of using just a single filter, we can use multiple filters as well. If we use multiple filters, the output dimension will change. For example, if we use 2 filters, we would have a **4x4x2** output image instead of **4x4**.

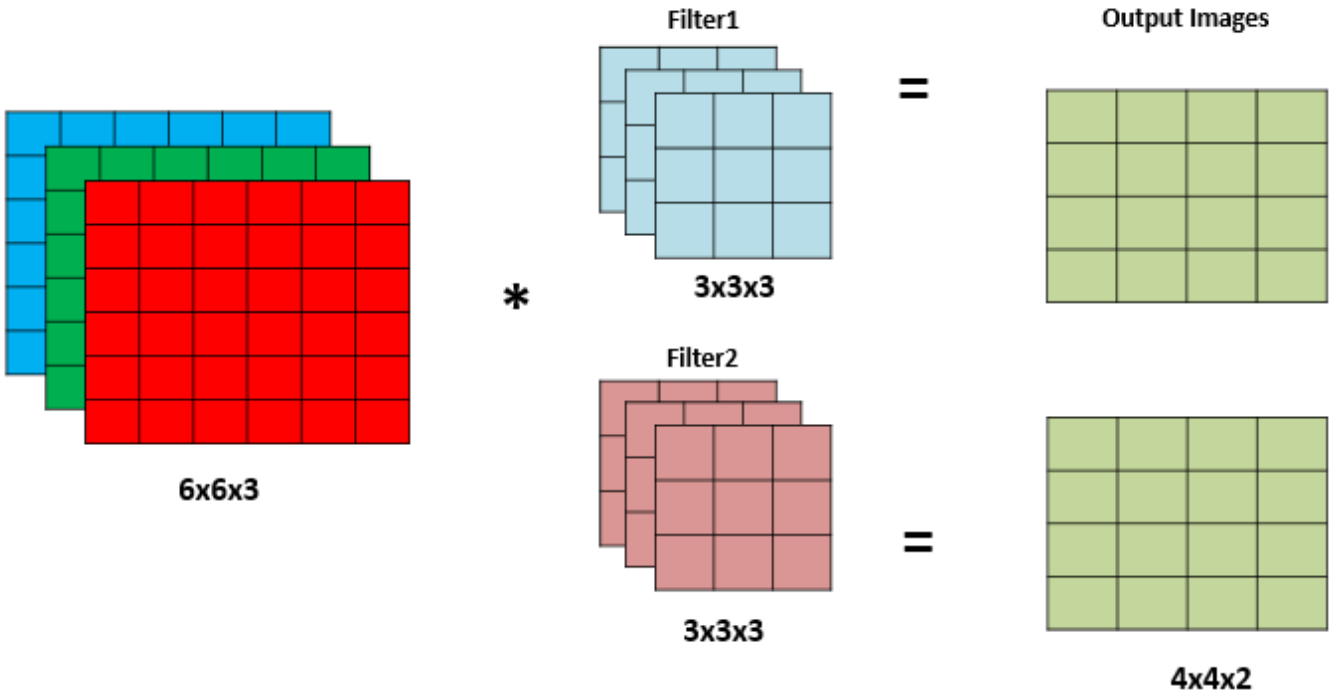


Figure 19: Convolution using multiple filters

We can generalize this and deduct the size of the output image:

- Input image size = n x n x Nc (Nc is number of channels)
- Filter size = f x f x Nc
- Number of filters = Nf
- Padding = p
- Stride = s

Then the output image size will be

$$\left(\frac{(n+2p-f)}{s}+1\right) \text{X} \left(\frac{(n+2p-f)}{s}+1\right) \text{X Nf}$$

2. Activation function

Once we get an output after convolving over the entire image using a filter, we add a bias term to those outputs and then we apply an activation function to generate activations. *This is one layer of a convolutional network*. Recall that the equation for one forward pass is given by:

$$z^{(1)} = w^{(1)} * a^{(0)} + b^{(1)}$$

$$a^{(1)} = g(z^{(1)})$$

$w^{(1)}$: represents the filter

$a^{(0)}$: represents the input image

$b^{(1)}$: represents the bias term

g() : represents the activation function, it's often the ReLU function.

$a^{(1)}$: represents the next input image

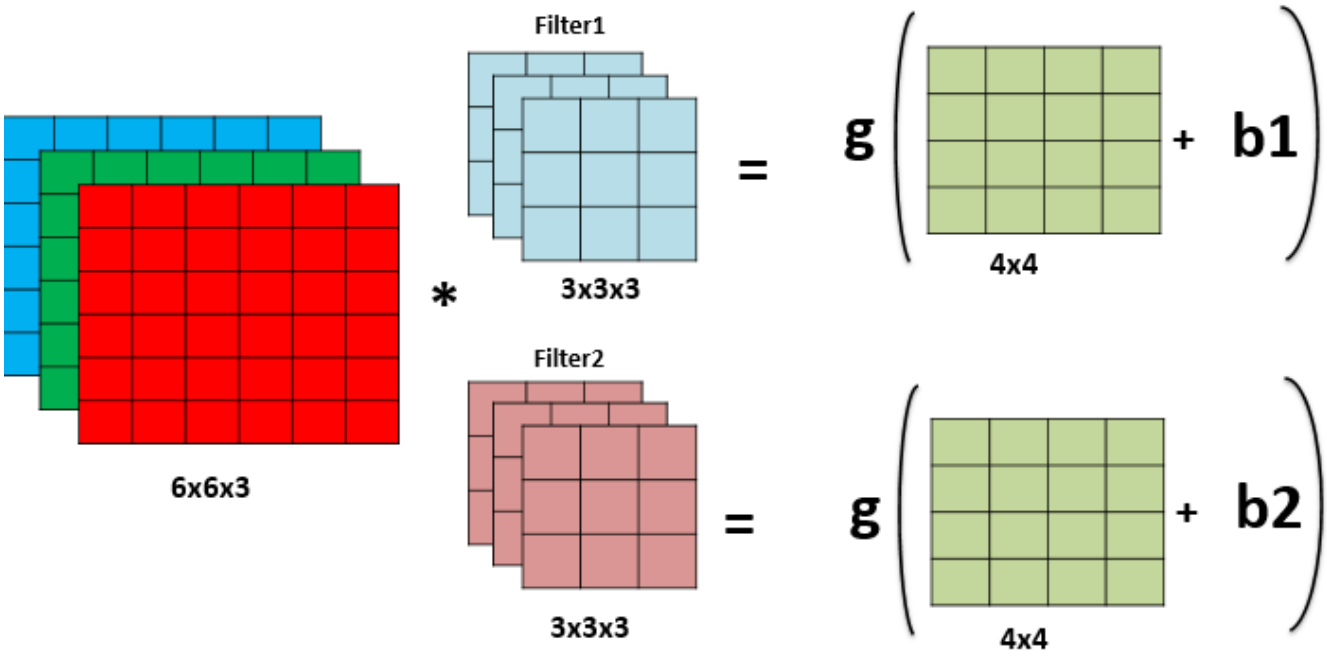


Figure 20: activation function and bias term

3. Pooling layer

Another kind of important layer in convolutional neural networks are pooling layers which apply the pooling operation. Pooling in general consists of transforming feature maps to smaller, more aggregated feature maps. The most common kind of pooling is **max-pooling**, which takes non-overlapping regions of a given size and outputs the largest activation in that region, e.g. splitting the input map into sections of size k x k (2 x 2 for example), choosing the maximum value and creating an output map consisting of those values. A pooling layer thus pools the input from a preceding convolutional layer, preserving the number of feature maps. The rationale behind pooling is to shrink the spatial dimensionality

in the network while preserving spatial information, using the strong correlation between data points that are close to each other. As with convolutions, pooling layers can use stride to achieve even greater aggregation.

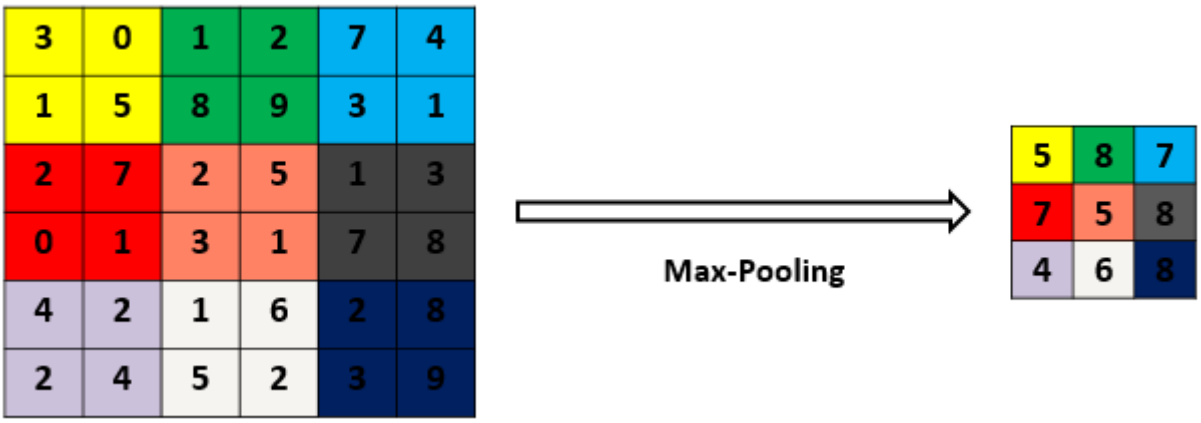


Figure 21: Max-pooling with 2x2 filter and stride =2

There are other types of pooling: The average pooling, the min pooling, the global pooling…etc., that takes the mean value of the matrix. But the max pooling has proven effective, because it extracts the most important features like edges.

4. Fully connected

After several layers of convolution and max-pooling, the output obtained is flattered and the high-level reasoning in the neural network is done via **fully connected** layers. The neurons in a fully connected layer have connections to all of the outputs in the previous layer.

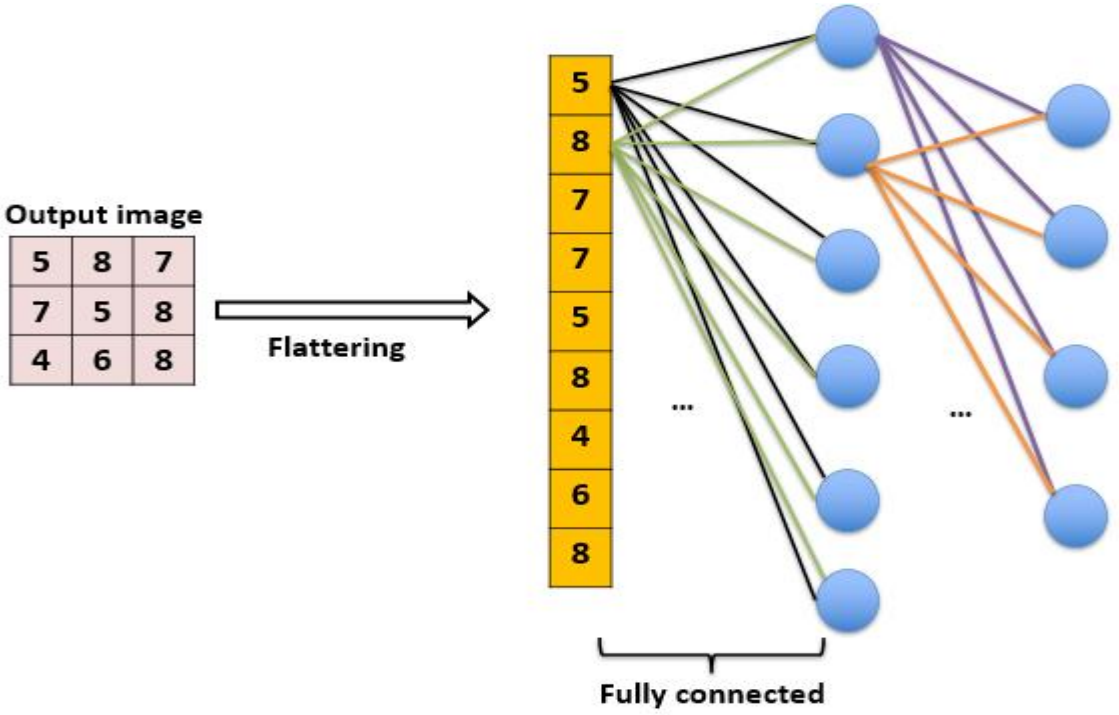


Figure 22: Fully connected layer

5. Loss function

Loss function helps in optimizing the parameters of the neural networks. Our objective is to minimize the loss for a neural network by optimizing its parameters(weights).
The loss is calculated using **loss function** by matching the target(actual) value and predicted value by a neural network.

Various loss functions suitable for different tasks can be used there. The "SoftMax" function calculates the probability distribution on the output classes.

C. Summary

The CNN architecture is composed of 4 main layers:

1. **The convolution**: the first and the most important layer hence the name of CNN, It has three parameters:
   - Number of kernels
   - Stride
   - Padding

   that allows to size the volume of the convolutional layer.

   The role of this layer is to extract the characteristics of the input image
2. **The max-pooling layer**: it's role is to reduce the size of the output image of the convolutional layer since it's extract only the maximum number in each matrix block
3. **The Fully connected layer**: the output matrix is flattered and entered in the neural network to treat it and then classify it.
4. **Loss function:** it's a function applied on the last output to determine in which class our input image belongs to.

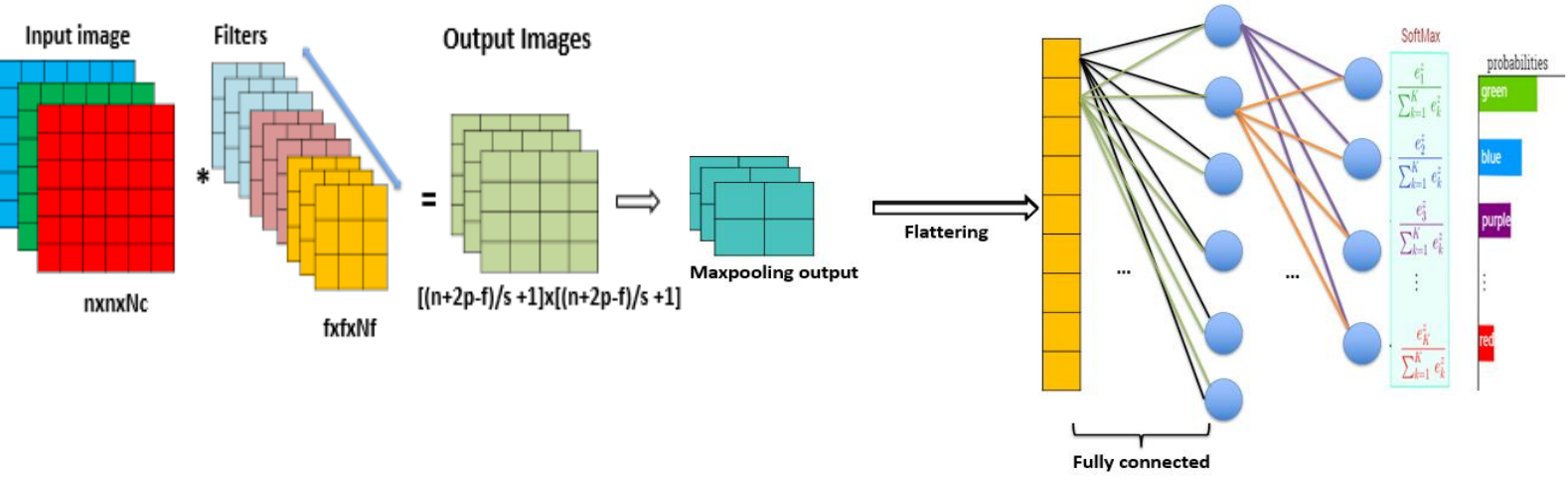


Figure 23: Recapitulative

IV. PROJECT IMPLEMENTATION

A. Installation and project requirements

1. Dataset

CIFAR-10 Dataset used in this paper consists of 60000x32 x 32 color images divided in 10 classes, with 6000 images in each class. There are 50000 training images and 10000 test images.

The chosen CIFAR-10 dataset is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5,000 images from each class.

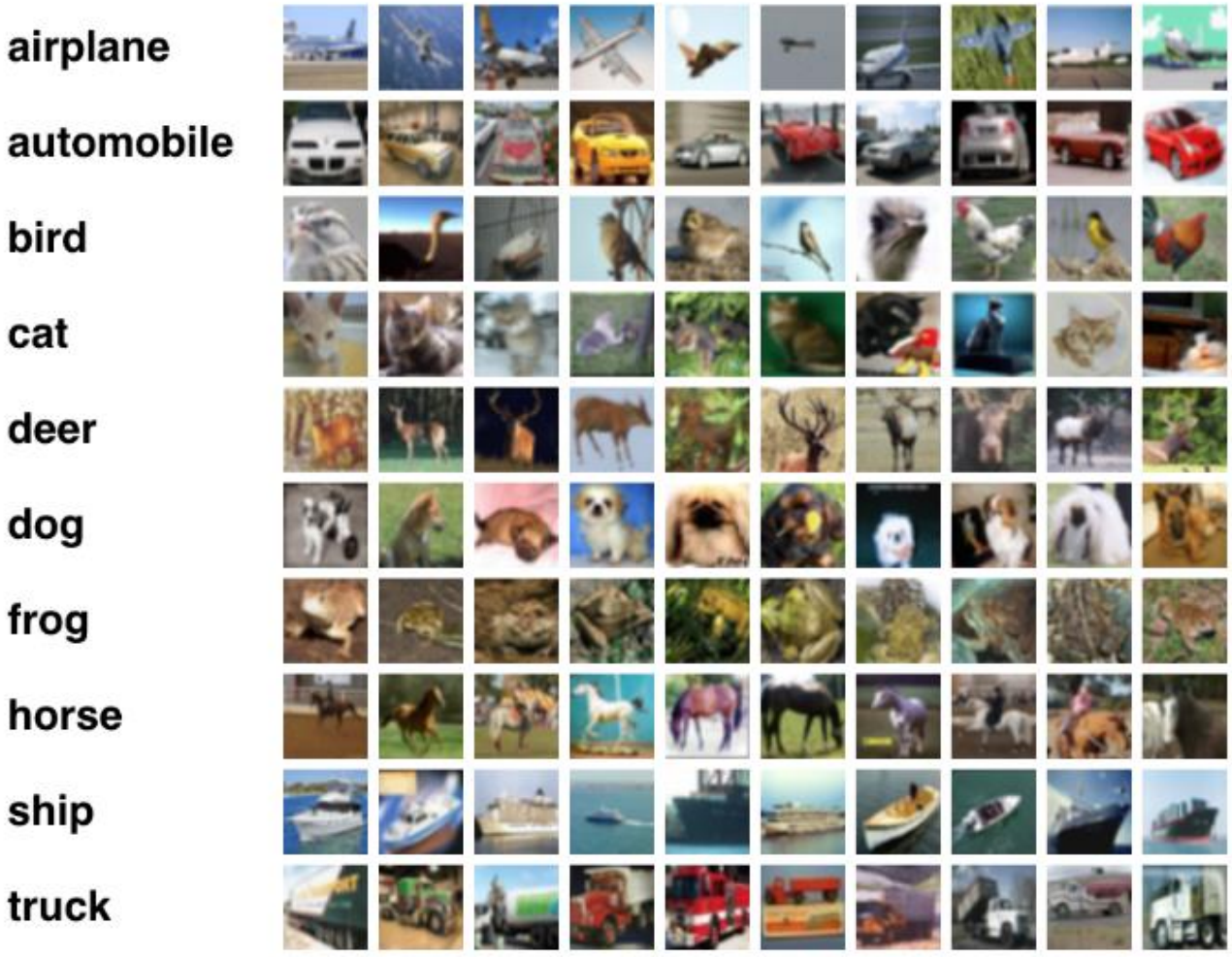


Figure 24 : CIFAR-10 Dataset

Machine learning is a complex discipline. But implementing machine learning models is far less daunting and difficult than it used to be, thanks to machine learning frameworks that ease the process of acquiring data, training models, serving predictions, and refining future results.

The following sections presents the frameworks and libraries used in our model.

2. Tensorflow

Created by the Google Brain team, TensorFlow is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning (aka neural networking) models and algorithms and makes them useful by way of a common metaphor. It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++.

TensorFlow can train and run deep neural networks for handwritten digit classification, image recognition, word embeddings, recurrent neural networks, sequence-to-sequence models for machine translation, natural language processing, and PDE (partial differential equation) based simulations. Best of all, TensorFlow supports production prediction at scale, with the same models used for training.

3. Keras

Keras is one of the leading high-level neural networks APIs. It is written in Python and supports multiple back-end neural network computation engines.

Keras was created to be user friendly, modular, easy to extend, and to work with Python. The API was “designed for human beings, not machines,” and “follows best practices for reducing cognitive load.” Neural layers, cost functions, optimizers, initialization schemes, activation functions, and regularization schemes are all standalone modules that we can combine to create new models. Using Keras , new modules are simple to add, as new classes and functions. Models are defined in Python code, not separate model configuration files.

B. Code overview

In this section part we will try to explain line by line our code and how it works. This part is divided into 4 parts:

- Loading and displaying cifar-10 dataset
- Pre-processing dataset
- Building the model
- Testing the model

1. Loading and displaying cifar-10 dataset

First of all we start by importing libraries we need (already explained in the section VI.A)

```
%matplotlib inline
import tensorflow as tf
import keras
import numpy as np
import matplotlib.pyplot as plt
import cv2
print(tf.__version__)
print(keras.__version__)
```

```
Using TensorFlow backend.
2.2.0-rc3
2.3.1
```

Then we import and upload the cifar-10 dataset using the code bellow:

```
from keras.datasets import cifar10
(x_train1, y_train1), (x_test1, y_test1) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 2s 0us/step
```

As mentioned before our dataset is be divided into two parts: train dataset and test dataset, their shapes can be shown using this code:

```
print("Data Shapes")
print("Train (x,y) => ",end = " ");print(x_train1.shape,y_train1.shape)
print("Test  (x,y) => ",end = " ");print(x_test1.shape,y_test1.shape)
```

```
Data Shapes
Train (x,y) =>  (50000, 32, 32, 3) (50000, 1)
Test  (x,y) =>  (10000, 32, 32, 3) (10000, 1)
```

The result shows that we have 50000 pictures in the train dataset and 10000 in the test data. To visualize the data, we will display 36 random images using the code below:

```
NUM_CLASSES = 10
imageDimesions = (32,32,3)

#dictionary of classes
code = {'plane':0, 'car':1, 'bird':2, 'cat':3, 'deer':4, 'dog':5, 'frog':6, 'horse':7, 'ship':8, 'truck':9}

#function getcode to get the value of key from the dictionnary
def getcode (n):
    for i,j in code.items() :
        if j == n :
            return i

# display 36 random images from the train dataset
plt.figure(figsize=(10,10))
for n, i in enumerate(list(np.random.randint(0,len(x_train1),36))) :
    plt.subplot(6,6,n+1)
    plt.imshow(x_train1[i])
    plt.axis('off')
    plt.title(getcode(y_train1[i]))
```

The result shows 36 labeled images:

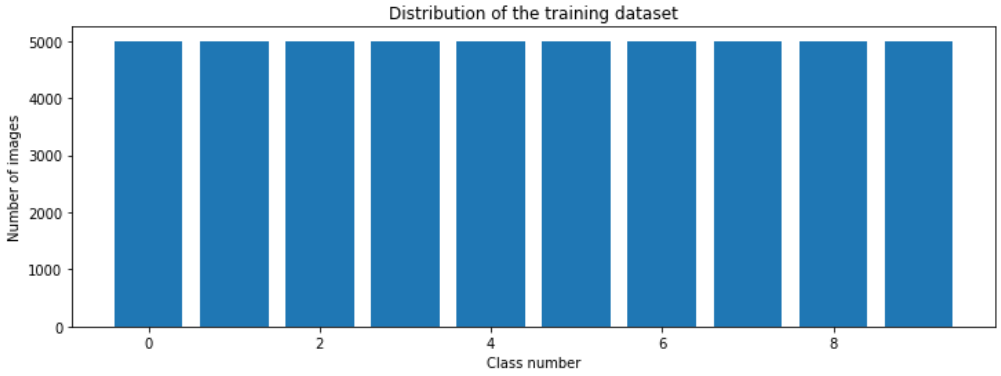
Now we know that we have 50000 pictures in the train dataset , moreover we have 10 classes, so in order to know how many samples in each class and display them we use this codes :

```
#calculate the number of samples for each category
num_of_samples = []
for x in range(0,10):
    num_of_samples.append((len(np.where(y_train1==x)[0])))
num_of_samples
```

```
[5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000]
```

```
# display a bar chart showing num_of_samples
print(num_of_samples)
plt.figure(figsize=(12, 4))
plt.bar(range(0, NUM_CLASSES), num_of_samples)
plt.title("Distribution of the training dataset")
plt.xlabel("Class number")
plt.ylabel("Number of images")
plt.show()
```

```
[5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000]
```



2. Preprocessing the dataset

Our data is composed from images which is understood by the machine as being matrix of pixels which are presented by numbers between 0 and 255 so we will preprocess this data by making a normalization between 0 and 1 instead of 0 and 255 using this code :

```
from keras.utils import to_categorical
#normalize values between 0 and 1 instead of 0 to 255
def preprocessing(img):
    img = (img/255)
    return img
#preprocess all images
x_train2=np.array(list(map(preprocessing,x_train1)))
x_test2=np.array(list(map(preprocessing,x_test1)))
```

After that we will convert the output class vector (integers) to binary class matrix using to_categorical function of keras.utils library.

```
# convert the output data to categirical one
y_train2= to_categorical(y_train1,NUM_CLASSES)
y_test2 = to_categorical(y_test1,NUM_CLASSES)
print ("y-train1 :")
print(y_train1)
print ("y-train2 :")
print(y_train2)
```

```
y-train1 :
[[6]
 [9]
 [9]
 ...
 [9]
 [1]
 [1]]
y-train2 :
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 0. 0. 1.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]]
```

To better understand the role of to_categorical function, we will take an example:

Let's consider the output vector **Y**:

6	**4**	**9**	**5**	**4**	**6**	**…**	**3**	**2**	**3**	**6**	**1**

When we apply to_categorical function to this vector we will obtain for each number, an array of size (1,10) and countain 1 in the position of each cell as below:

0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
0	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
					…				
0	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

The first array represents the first cell in the Y output vector and so on…

3. Developing the model

In order to build the model, we first need to import some functions:

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten , Dense, Activation,Dropout,BatchNormalization
```

The model contains few techniques which were proposed recently and has become a general norm these days in convolutional neural networks.

Conv2D: This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

Activation: This method is the activation function that is applied to the output (explained in the section II.A)

BatchNormalization : This layer will transform inputs so that they are standardized, meaning that they will have a mean of zero and a standard deviation of one.

maxPooling: this function represents the maxpooling layer

Dropout: this function is added to avoid the overfitting of the model.

Flatten: used to flatten the output matrix.

Dense: implements the operation $\mathbf{output} = \mathbf{activation}(\mathbf{dot}(\mathbf{input}, \mathbf{kernel}) + \mathbf{bias}$ where activation is the element-wise activation function passed as the activation argument , kernel is a weights matrix created by the layer , and bias is a bias vector created by the layer.

```
def make_model():
    model = Sequential()
    model.add(Conv2D(32, (3,3), padding='same',  input_shape=(32,32,3)))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(32, (3,3), padding='same',  input_shape=(32,32,3)))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(64, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.3))

    model.add(Conv2D(128, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(128, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(128, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.4))
```

```
    model.add(Conv2D(256, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(256, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Conv2D(256, (3,3), padding='same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.5))

    model.add(Flatten())
    model.add(Dense(300))
    model.add(Dense(100))
    model.add(Dense(NUM_CLASSES, activation='softmax'))
    return model
```

Then we can check those layers using summary function:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
activation_3 (Activation)    (None, 16, 16, 64)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
activation_4 (Activation)    (None, 16, 16, 64)        0
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
activation_5 (Activation)    (None, 8, 8, 128)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
activation_6 (Activation)    (None, 8, 8, 128)         0
_________________________________________________________________
batch_normalization_6 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
activation_7 (Activation)    (None, 8, 8, 128)         0
_________________________________________________________________
```

batch_normalization_7 (Batch	(None, 8, 8, 128)	512
max_pooling2d_3 (MaxPooling2	(None, 4, 4, 128)	0
dropout_3 (Dropout)	(None, 4, 4, 128)	0
conv2d_8 (Conv2D)	(None, 4, 4, 256)	295168
activation_8 (Activation)	(None, 4, 4, 256)	0
batch_normalization_8 (Batch	(None, 4, 4, 256)	1024
conv2d_9 (Conv2D)	(None, 4, 4, 256)	590080
activation_9 (Activation)	(None, 4, 4, 256)	0
batch_normalization_9 (Batch	(None, 4, 4, 256)	1024
conv2d_10 (Conv2D)	(None, 4, 4, 256)	590080
activation_10 (Activation)	(None, 4, 4, 256)	0
batch_normalization_10 (Batc	(None, 4, 4, 256)	1024
max_pooling2d_4 (MaxPooling2	(None, 2, 2, 256)	0
dropout_4 (Dropout)	(None, 2, 2, 256)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 300)	307500
dense_2 (Dense)	(None, 100)	30100
dense_3 (Dense)	(None, 10)	1010

Total params: 2,253,906
Trainable params: 2,251,218
Non-trainable params: 2,688

After that we define the model and then we prepare it for fitting: we define the loss instance, the optimizer (we used adamax optimizer) and the metrics that we want to see during the training:

```
[ ] INIT_LR = 3e-3
    EPOCHS = 20
```

```
[ ] model = make_model()
    model.compile(
        loss='categorical_crossentropy',
        optimizer=keras.optimizers.adamax(lr=INIT_LR),
        metrics=['accuracy']
    )
```

Finally, we train the model

```
[ ] model.fit(
        x_train2,y_train2,
        epochs=EPOCHS,
        validation_data=(x_test2, y_test2),
        shuffle=True,
        verbose=1,
        initial_epoch= 0
    )
```

4. Testing the model

After the model training, we will use the code below to test its performance:

```
y_pred_test = model.predict_proba(x_test2)
y_pred_test_classes = np.argmax(y_pred_test, axis=1)
y_pred_test_max_probas = np.max(y_pred_test, axis=1)

score =model.evaluate(x_test2,y_test2,verbose=0)
print('Test Score:',score[0])
print('Test Accuracy:',score[1])

cols = 8
rows = 2
fig = plt.figure(figsize=(2 * cols - 1, 3 * rows - 1))
for i in range(cols):
    for j in range(rows):
        random_index = np.random.randint(0, len(y_test1))
        ax = fig.add_subplot(rows, cols, i * rows + j + 1)
        ax.grid('off')
        ax.axis('off')
        ax.imshow(x_test1[random_index, :])
        pred_label =  getcode(y_pred_test_classes[random_index])
        pred_proba = y_pred_test_max_probas[random_index]
        true_label = getcode(y_test1[random_index, 0])
        ax.set_title("pred: {}\nscore: {:.3}\ntrue: {}".format(
               pred_label, pred_proba, true_label
        ))
plt.show()
```

Finally, we display and plot the confusion matrix using the code below:

```
yPred = model.predict_classes(x_test2)
yTest_original=np.argmax(y_test2,axis=1)

class_names = np.unique(y_test1)

from sklearn.metrics import confusion_matrix
from pandas import DataFrame
from seaborn import heatmap

cnf_matrix = confusion_matrix(y_true=yTest_original, y_pred=yPred, normalize='true')
plot_parameters = DataFrame(cnf_matrix, index = code.keys() ,columns = code.keys())
plt.figure(figsize = (15,7))
heatmap(plot_parameters, annot=True)
```

C. Results and explanation

1. Performance measures

When training a machine learning model, one of the main things that we want to avoid would be overfitting. This is the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably.

To find out if their model is overfitting, data scientists use a technique called cross-validation, where they split their data into two parts - the training set, and the validation set. The training set is used to train the model, while the validation set is only used to evaluate the model's performance.

Metrics on the training set let us see how our model is progressing in terms of its training, but its metrics on the validation set that let us get a measure of the quality of the model - how well it's able to make new predictions based on data it hasn't seen before.

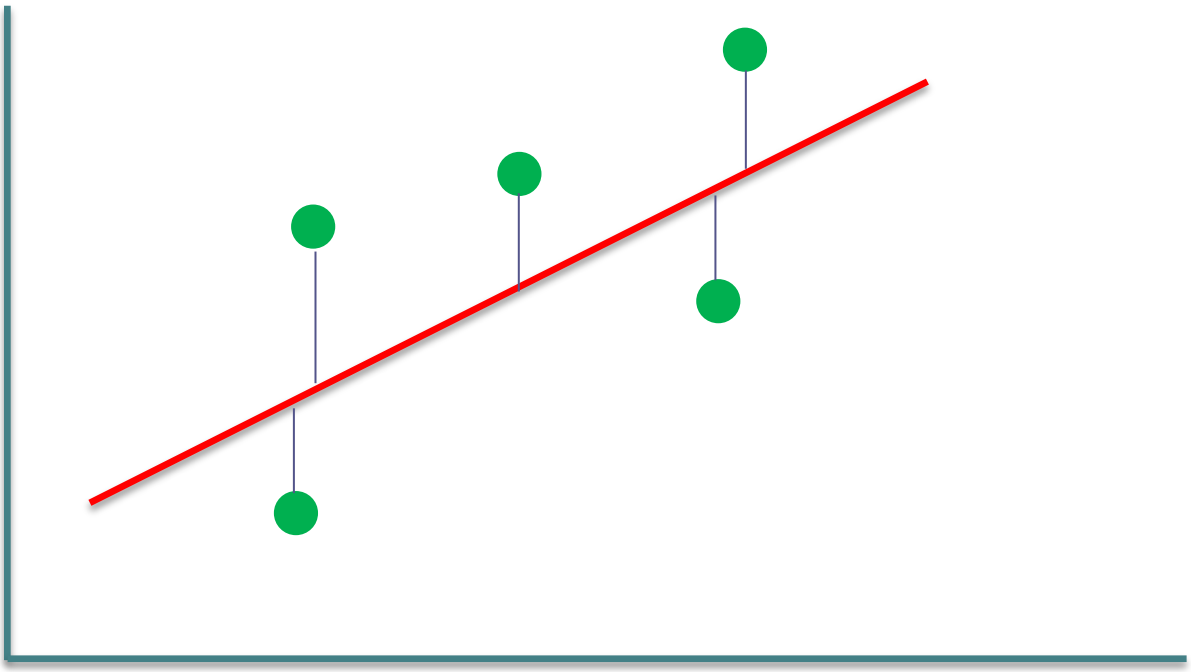
The sections below present the metrics we used to test our data:

⇨ **Accuracy:**

The accuracy of a machine learning classification algorithm is one way to measure how often the algorithm classifies a data point correctly. Accuracy is the number of correctly predicted data points out of all the data points. More formally, it is defined as the number of true positives and true negatives divided by the number of true positives, true negatives, false positives, and false negatives.

- ⇨ **MSE**:

To better understand the MSE metric we will take these green points and our goal is to be very close to the red line :



So, we can say that the squared distance between the green points and the red line represents the MSE metric and it must be too close to 0 to have a good model.

- ⇨ **Confusion Matrix**

it is a performance measurement for machine learning classification problem where output can be two or more classes. In fact , it is a table that is often used to describe the performance of a classification model (or “classifier”) on a set of test data for which the true values are known. shows the ways in which your classification model is confused when it makes predictions.

2. Model Result

In the section above we create a code that takes the history of training and then plots the training and testing accuracy, loss and MSE. the figures below show the obtained results:

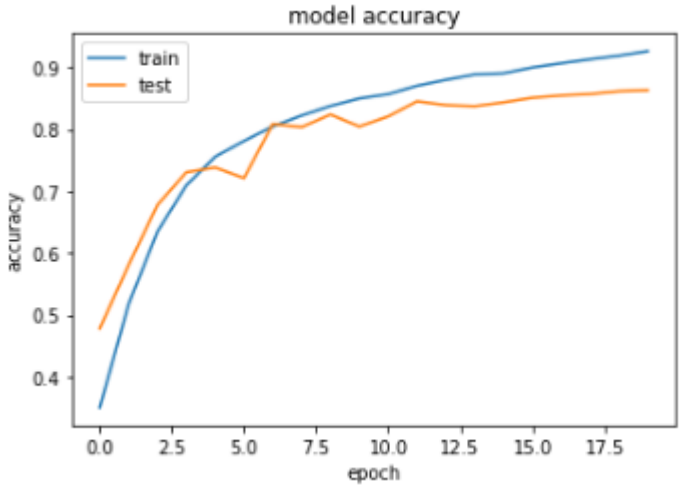


Figure 25: model accuracy

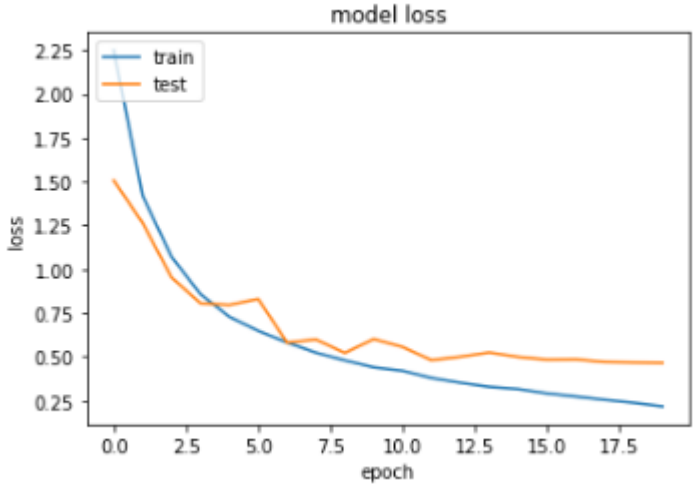


Figure 26: model loss

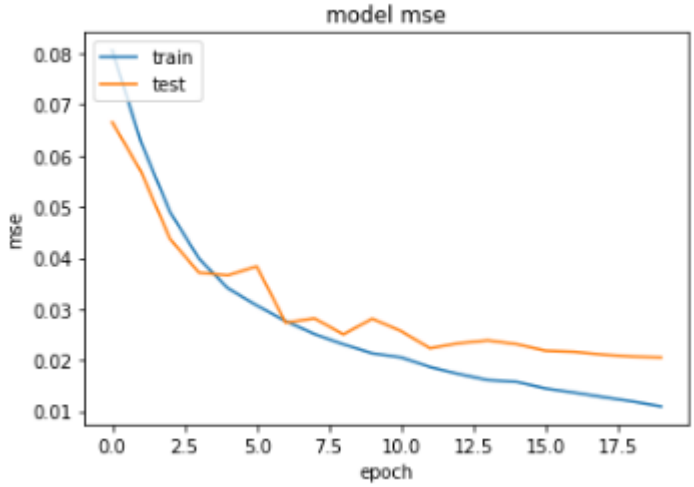


Figure 27: model mse

The training and testing accuracy increases with the number of epochs, this reflects that in each epochs the model learns more and more information, which enhance the performance of the model.

Similarly, the train loss and the test loss are decreasing when the number of epochs increases and that reflects that our model gets better and better every time we train.

MSE metric also decrease when the number of epochs increases.

To evaluate the model, we create a test code that give that result:

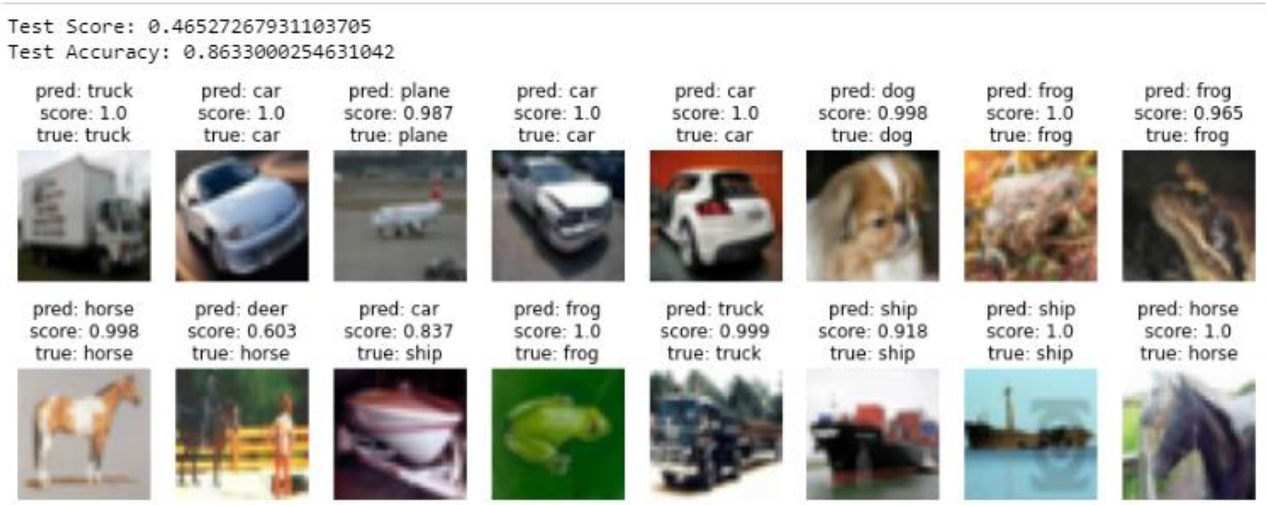


Figure 28: image classification made by the model

We remark that our model achieved the accuracy of 86,3% which is a good percentage to classify images.

Finally, the confusion matrix below also shows that the True positive and the false negative values are very high and the diagonal of the matrix is highlighted which mean that the model is considered as a good model.

Figure 29: confusion matrix

we got this result after several training of different models that we can summarize in this table:

Architecture of the model					Epoch number	Train accuracy	Validation accuracy	MSE
Conv layer	Pooling layer	Normalization layer	Dropout	Fully-connected				
10	**4**	**10**	**4**	**3**	**20**	**92%**	**86.3%**	**0.0205**
10	**4**	**10**	**4**	**3**	**50**	**98%**	**87%**	**0.02**
10	**4**	**10**	**4**	**3**	**10**	**75%**	**77%**	**0.0313**
8	**4**	**8**	**4**	**1**	**20**	**95%**	**85%**	**0.0226**
8	**4**	**0**	**4**	**1**	**20**	**80%**	**77%**	**0.0320**
8	**4**	**0**	**0**	**1**	**20**	**75%**	**55%**	**0.0510**

we can conclude from these results the following points:

- The normalization layer is very important since it enhanced the accuracy from 77% to 85% that means we gained 8% of the accuracy
- When the number of epochs increases the accuracy increases also, but at a given value of epochs the accuracy doesn't change too much as it’s mentioned in the table.
- The dropout function avoids the overfitting because when we didn’t use it, the training accuracy was good and far from the test accuracy that means that the model is overfitting
- Convolutional layers give good results

Finally, we can say that CNN in general gives good accuracy and can well classify images compared to other architectures.

D. Project deployment

In order to test its performance, we created a little application that can take new images and predict their class using the file cifar_model.h5 in which we saved the weight of our model. The source code of the application is link in the annex section.

Here is some result of images predicted by our model . Note that our model hasn’t seen these images before:

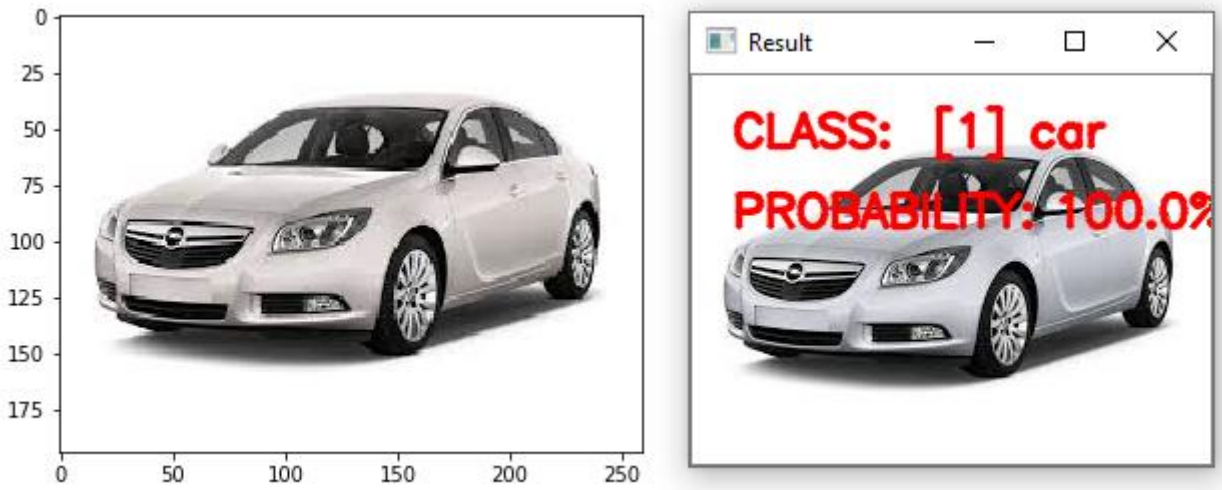
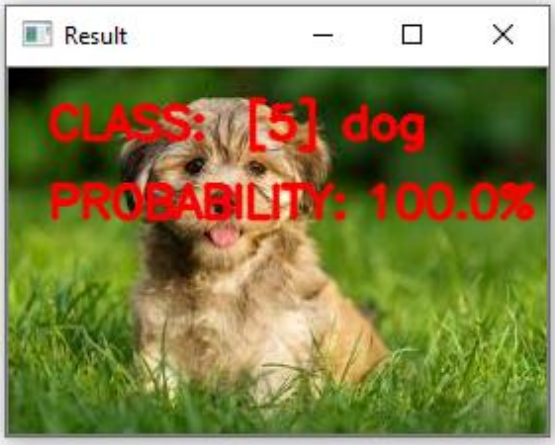
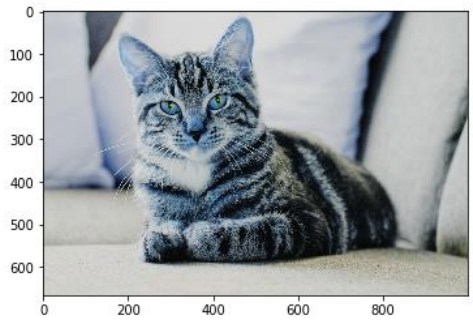


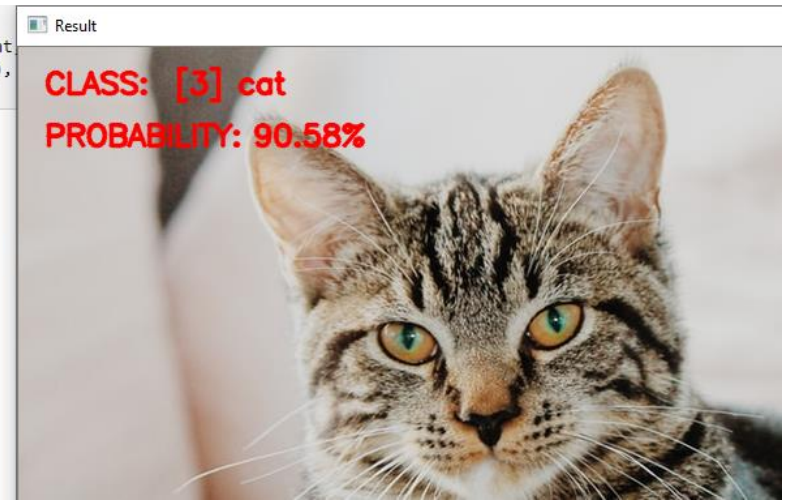
Figure 30: new images predicted by the model




```
img = img.reshape(1, 32, 32, 3)
cv2.putText(imgOrignal, "CLASS: " , (20, 35), font
cv2.putText(imgOrignal, "PROBABILITY: ", (20, 75),
cv2.LINE_AA
```

Out[34]: 16





We conclude that our model can even predict images that hasn't seen before with very high probability which reflect the power of CNN models in classifying images.

V. Conclusion

In this project we discussed the basic notions of neural networks in general and convolutional neural networks in particular. We have introduced these convolutional neural networks by presenting the different types of layers used in the classification: the convolutional layer, the rectification layer, the pooling layer and the fully connected layer. We also talked about the regularization methods (dropout) used to avoid the problem of overfitting.

Regarding the implementation, we presented in this paper a classification approach based on convolutional neural networks, for this we used a model with a specified architecture and we showed the different results obtained in terms of precision and error. The analysis of the results found has shown that the CNN is a very powerful way to classify image since it gives very high accuracy (86%) and low error.

We encountered some problems in the implementation phase, the use of a CPU made the execution time too expensive. In order to resolve this problem, we deployed our model and we trained it on a TPU instead of a CPU using google colab platform that offers free resources to the researchers in order to train their model.

This project was an opportunity for us to learn more about deep learning and deepen our knowledge in machine learning in general and CNN in particular. In fact, we learned how to build good model, how to test its performance and how to analyses its results.

WEBOGRAPHY

https://fr.wikipedia.org/wiki/R%C3%A9seau_neuronal_convolutif

https://www.tensorflow.org/tutorials/images/cnn

https://www.tensorflow.org/api_docs/python/tf/keras/layers/

ANNEXE

Link to the source code of the model:

https://github.com/Bougrine-Imane/CNN_Project/blob/master/CNN_Cifar_10-VF.ipynb

Link to the source code of the application:

https://github.com/ysfelkantri/CNN_CIFAR10/blob/master/app.ipynb

link to the demonstration video of the application

https://www.youtube.com/watch?v=IYQTvlx_RYk&feature=youtu.be