المدرسة العليا للتكنولوجيا - اكادير
ÉCOLE SUPÉRIEURE DE TECHNOLOGIE D'AGADIR

HANDWRITTEN DIGITS RECOGNIZER

End of Studies Project Report

Created by

Miss. KAOUTAR SOUGRATI

Mr. AYOUB LAMFADLI

Mr. HAMZA AIT BOURHIM

Supervised by

Dr. MUSTAPHA AMROUCH

Defended on May 27th in front of the jury

Dr. MUSTAPHA AMROUCH

Dr. ANSARI YOUNES

2020/2021

ACKNOWLEDGEMENTS

To begin with, we would like to express our sincere appreciation to the faculty and administration of the Higher Institute of Technology for their services and efforts made during the last two years.

We are thankful to Dr. MUSTAPHA AMROUCH for being our supervisor for this project, and for having guided and advised us through it. He has been very supportive throughout the whole course of our project.

We are also thankful to Mr. Michael Nielsen for writing and publishing his online book named Neural Networks and Deep Learning. It helped us understand the logic and implement an important part of our project. Without forgetting to thank the National Institute of Standards and Technology for providing the dataset of labeled handwritten digits worked with.

Finally, we also extend our sincere thanks and our deep gratitude to all those who have supported and encouraged us.

CONTENTS

LIST OF FIGURES

ABSTRACT

In this document, we are going to report the steps and results of the journey we had towards accomplishing the implementation of an algorithm that allows the computer to recognize handwritten digits input. The creation of this software was divided into two parts. The first one is the interface part and the second one is the Machine Learning part. It uses an Artificial Neural Network algorithm to learn from the Modified National Institute of Science and Technology dataset of labeled handwritten digits. Then, it presents an interface for the user to enter their own digit, and then displays the result. This document will show the steps followed to augment the precision of the result produced by the back-end part and the creation and adaptation of the front-end interface to comply precisely with the recognition mechanism.

LIST OF ACRONYMS AND ABBREVIATIONS

ML: Machine Learning

AI: Artificial Intelligence

NN: Neural Network

CNN: Convolutional Neural Network

HDR: Handwritten Digit Recognition

MLP: Multilayer Perceptron

MNIST: Modified National Institute of Standards and Technology

FC: Fully connected

LSTM: Long Short Term Memory

GUI: Graphical User Interface

CHAPTER 1: GENERAL CONTEXT AND SPECIFICATION

1 INTRODUCTION

Artificial Intelligence (AI) is the general term that refers to the simulation of human intelligence in machines that are programmed to think like humans and mimic their behavior.

AI is a very fast-evolving field. In the way to general artificial intelligence, researchers keep constantly creating new techniques to enhance “machines’ brains”. One of the most efficient subsets of AI is known as Machine learning[1].

Machine learning (ML) focuses on applications that learn from experience and improve their decision-making over time without being explicitly programmed. ML keeps growing in the modern world and has many different applications like image recognition, speech recognition, and medical diagnosis.

2 PROJECT PRESENTATION AND PROBLEM DEFINITION

In this project, we will be working specifically on **Handwritten Digit Recognition**. Handwritten digit recognition is the ability of computers to recognize human handwritten digits. It is a hard task for the machine because handwritten digits are not perfect and can be made with many different flavors. However, It is indeed a very important task that can be used in many applications such as robotics.

We will implement a handwritten digit recognition model in which we convert the image containing our digit into some type of data structures the computer would understand, and that by using a special type of deep neural networks called Convolutional Neural Networks.

3 OBJECTIVE

The main goal of this project is to implement an intelligent system that will be able to :

- Apply a Convolutional Neural Network[2] (CNN).
- Focus on LeNet-5 architecture[3].
- Differentiate between handwritten numbers.
- Make it easier to read and practice writing numbers.

4 PROBLEMATIC

In this digital world, almost any decent institution should have the ability to use the various forms of today's technology to enhance their services. A good example of the institutions in need of this would be our educational systems.

It is almost certain that the application of new technologies from the fields of Machine Learning can help improve the educational system and provide it with valuable resources and tools that can make the process of teaching easier and more efficient.

Therefore, we believe that our system will be a very demanded tool in primary schools to help children learn how to read and practice writing numbers, as well as learning them in different languages.

5 METHODOLOGY AND SOLUTION

In this project, we are going to implement a model to recognize handwritten digits using CNN. We will train the classifier by having it "looking" at thousands of labeled images of handwritten numbers. Then, We will assess the accuracy of the classifier using "test data" the model has never seen.

To reach this goal, we need to follow the following steps:

- Collecting Data
- Preparing Data
- Choosing a model
- Training the model
- Evaluating the model

6 PLANNING, PROGRESS AND ORGANIZATION OF THE PROJECT

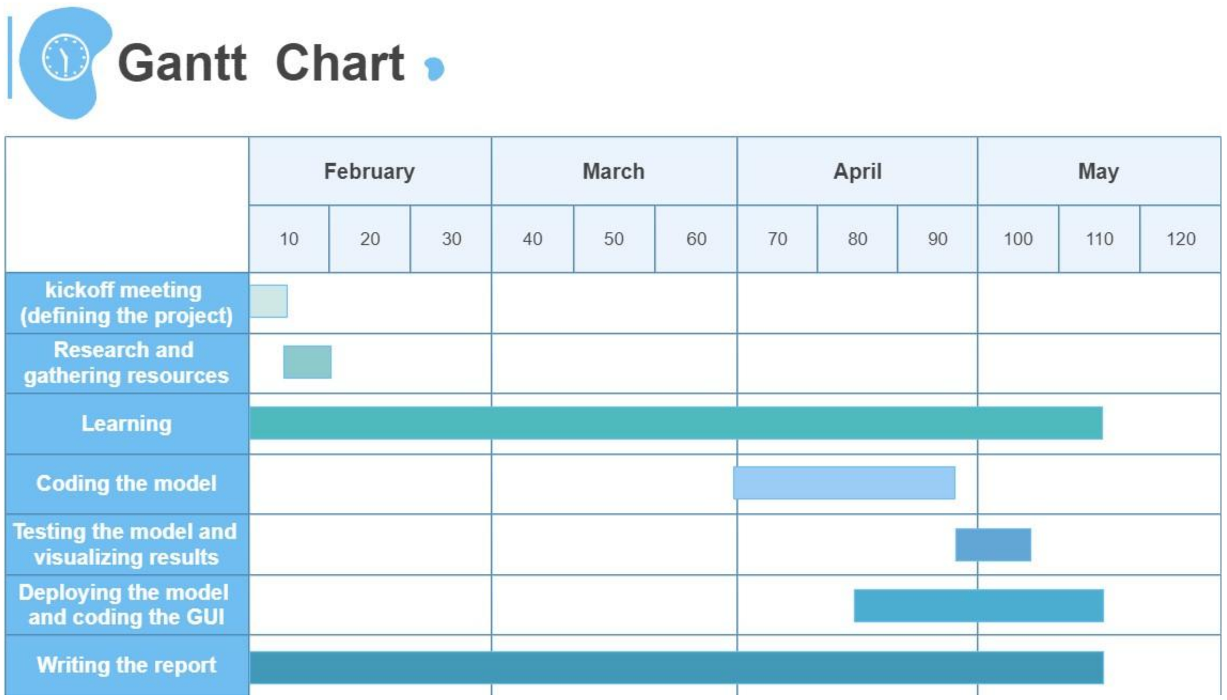


Figure 1: Gantt chart

7 TOOLS

7.1 Python

Python is an open-source popular programming language that can be used for a wide variety of applications. It includes high-level data structures, dynamic typing, dynamic binding, and many more features that make it useful for complex application development. Besides, Python code syntax uses English keywords, and that makes it easy for anyone to understand and get started with the language.

7.2 TensorFlow

TensorFlow is an open-source library for numerical computation and large-scale machine learning. It allows developers to create dataflow graphs—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array or tensor. 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.

7.3 Keras

Keras[4] is the high-level API of TensorFlow 2. an approachable, highly productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

7.4 Google Colab

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser and is especially well suited to machine learning, data analysis, and education. It does not require a setup and the notebooks that you create can be simultaneously edited by your team members and it supports many popular machine learning libraries that can be easily loaded in your notebook.

8 FEASIBILITY STUDY

On the technical level, our main concern will be coding the right algorithm for the classification. As for the input data, we will use a famous handwritten digits dataset assembled by the National Institute of Standards and Technology and arranged by Yann Lecun. It contains a training set of 60,000 examples and a testing set of 10,000 examples. They have been centered and size-normalized in fixed-size labeled images, each number in one image.

We will then use a step-by-step methodology to learn about different neural network algorithms and use one of the best algorithms for our specific application. Part of our work will consist of adapting the different codes for neural networks available to our program. Another part will consist of building an interface for the user to input a handwritten number.

In what concerns the resources, we will be programming on our laptops, using the Internet connection to download the data and search for the algorithms to adapt. Our work will not need any additional expenses. The work will be done by us (Kaoutar Sougrati, Ayoub Lamfadli and Hamza Ait Bourhim), under the supervision of Dr. Mustapha Amrouch. We estimate the time we will need to do it in four months.

9 CONCLUSION

This first chapter was devoted essentially to present the main problem and the solution methodology that we will follow. We also tried to mention the planning and the tools used in the project.

CHAPTER 2: NEURAL NETWORKS

1 INTRODUCTION

Neural Networks[5] are the main tool used in Machine learning. They deal with data through a process that mimics the way we think the human brain operates. A neural network contains layers of interconnected nodes. Each node is a perceptron[6]. A bunch of perceptrons forms a neural network. A “neuron” in a neural network is a mathematical function that collects and classifies information according to a specific architecture.

For instance, in our case, the neurons hold a number representing its activation. Hence, the neurons light up when their activation is a high number. The output should be the neuron within the highest activation.

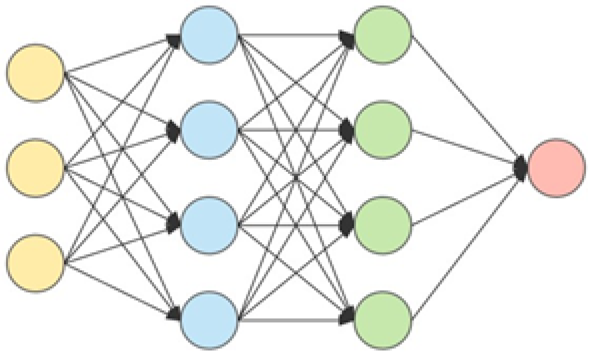


Figure 2 Neural Network

2 PERCEPTRON

2.1 PRESENTATION AND ARCHITECTURE

Before making our neural network for recognizing handwritten digits, we have to start by creating the smallest building block of neural networks which is a perceptron.

A perceptron is a neural network that takes as input either 0 or 1 (x1,x2,...) and produces a single binary output (The neural network we will use will not have binary output).

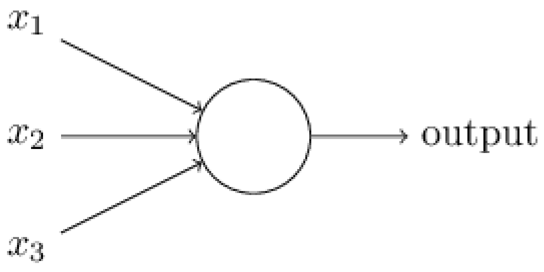


Figure 3 Perceptron

Each of the links has a **weight** w1,w2… that represents the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum ***∑wj*xj*** is less than or greater than a value called the **bias** this operation is known as the Heaviside function.

$$\text{if } \sum w_j * x_j \leq bias,\ output = 0$$

$$\text{if } \sum w_j * x_j > bias,\ output = 1$$

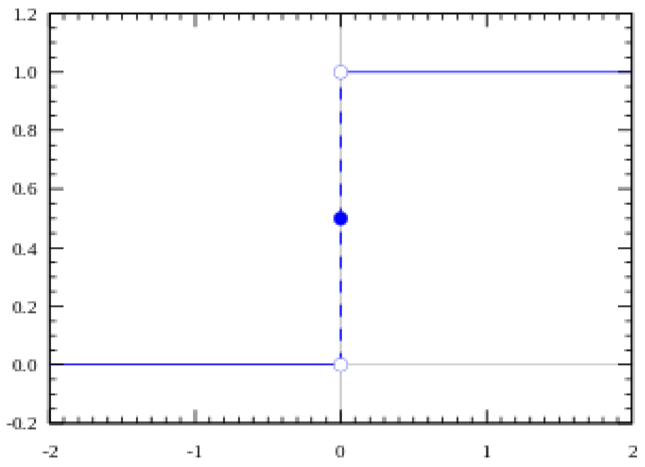


Figure 4 Curve of the Heaviside function

The perceptron then works by correcting the weights and biases depending on how close it was to the correct value while we are training it by giving different training data(inputs that we know the output to).

2.2 APPLICATIONS

Certain machine learning algorithms are more applicable in certain situations than others. So when is the single perceptron used?

The perceptron is an algorithm that is defined to be a **binary classifier**. This means that it can decide whether or not an input belongs to some class. And it can be used only when the dataset is linearly separable.

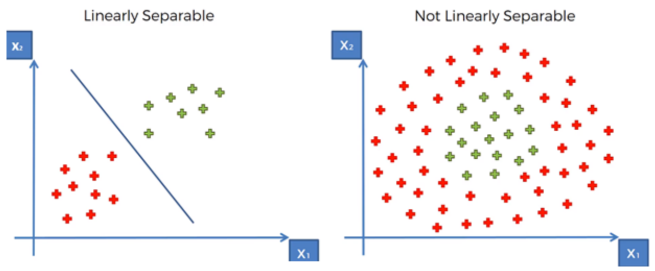


Figure 5 Linearly vs. non-linearly separable datasets

2.2.1 The neural representation of AND:

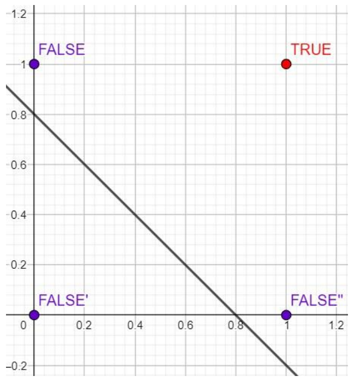
A simple way to understand how linear classification with perceptrons works is to apply it to the classical logic gates problem. We can define the AND gate as a perceptron that takes two binary inputs and outputs one binary "decision" about them.

Observation	x1	x2	AND(x1, x2)
1	0	0	FALSE
2	0	1	FALSE
3	1	0	FALSE
4	1	1	TRUE

Inputs : X(1, x1, x2)
Parameters : W(w0, w1, w2) (where w0 is the bias) and n = learning rate = 0.2
Output : Y = FALSE if $\sum w_j * x_j \leq 0$ and Y = TRUE if $\sum w_j * x_j > 0$
Update Step : wi' = wi + n*d*xi (where d = 1 if the line should be upper and d = -1 if it should be lower)

Step 1(initialization) :

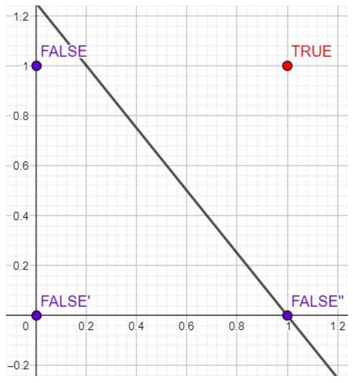
We start by giving random values to the parameters w0 = 0.8 , w1 = -1 , w2 = -1
Thus, $\sum w_j * x_j$ = -x1 -x2 + 0.8
Which we can use to find the equation of the following line :
x2 = -x1 + 0.8



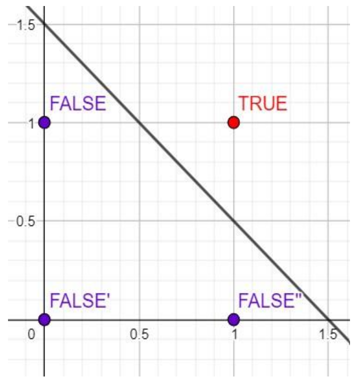
Step 2(learning) :

In this step, we should use the update step to update the parameters if there are any points that are not well classified.

- (0,0) and (1,1) points are well classified.
- For (0.1) it is misclassified so we need to update the parameters
  w0' = w0 + n*d*x0 = 0.8 + 0.2*(1)*1 = 1
  w1' = w1 + n*d*x1 = -1 + 0.2*(1)*0 = -1
  w2' = w2 + n*d*x2 = -1 + 0.2*(1)*1= -0.8
  The new equation therefore will be : **x2 = -1.25*x1 + 1.25**



- For (1,0) it is misclassified so we need to update the parameters
  w0' = w0 + n*d*x0 = 1 + 0.2*(1)*1 = 1.2
  w1' = w1 + n*d*x1 = -1 + 0.2*(1)*1 = -0.8
  w2' = w2 + n*d*x2 = -0.8 + 0.2*(1)*0= -0.8
  The new equation therefore will be : **x2 = -x1 + 1.5**



2.2.2 The neural representation of OR:

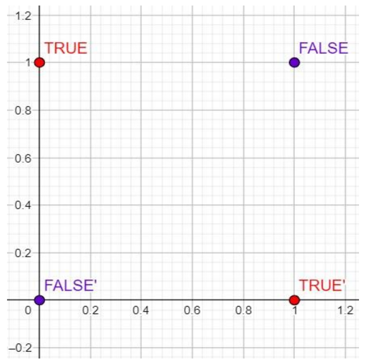
OR(x1, x2) will take a similar approach with the following calculation : **OR(x1, x2) = Ɵ(w1*x1 + w2*x2 + b)** (where x1,x2 are the inputs w1,w2 are the weights and b is the bias).

Observation	x1	x2	OR(x1.x2)
1	0	0	FALSE
2	0	1	TRUE
3	1	0	TRUE
4	1	1	TRUE

In this example, we are going to follow the exact same steps explained in the previous sample to adjust weights and biases (if needed).

2.2.3 The problem with XOR:

The XOR gate is a good example to understand the limitation of the single perceptron. The reason this cannot be solved with a single perceptron is that the XOR problem is **non-linear**. The XOR perceptron is associated with the following calculation: **XOR(x1, x2) = AND(NOT(AND(x1, x2)), OR(x1, x2))** (check the following sections for more information about multilayer perceptrons).



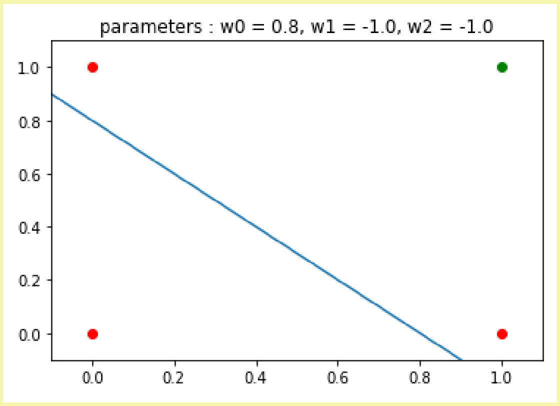
There is no line that can **separate** the TRUE dots from the FALSE dots!

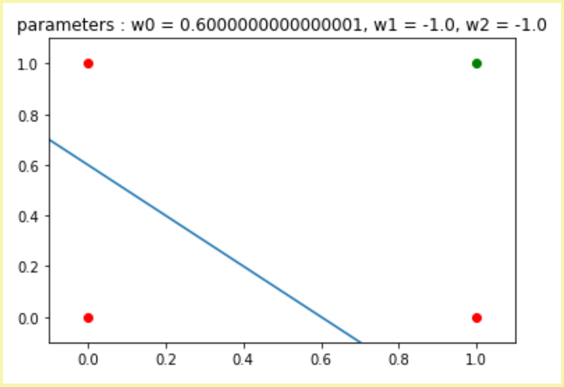
2.3 IMPLEMENTATIONS

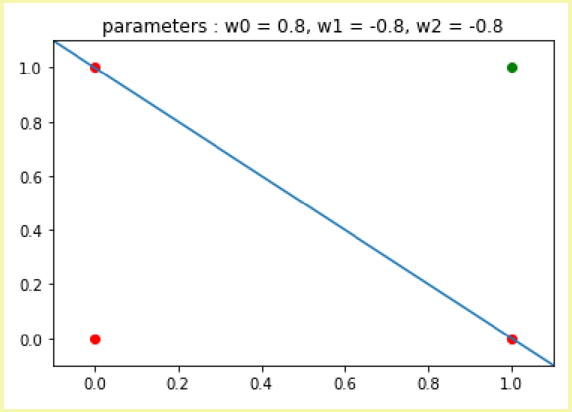
In this section, we tried to implement the code for the AND perceptron in two methods(with and without TensorFlow).

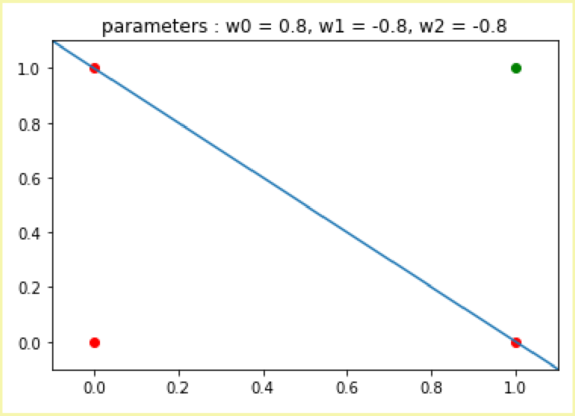
2.3.1 AND perceptron without using TensorFlow

```python
# importing numpy library
import numpy
from pylab import ylim, plot
from matplotlib import pyplot as plt

# define perceptron function
def perceptron(W, X):
  if numpy.dot(W, X) > 0:
    return 1
  else:
    return 0

#data visualization
def visualize(W, X):
  fig, ax = plt.subplots()
  X = numpy.arange(-0.2, 1.2, 0.1)
  ax.scatter(0, 0, color="r")
  ax.scatter(0, 1, color="r")
  ax.scatter(1, 0, color="r")
  ax.scatter(1, 1, color="g")
  ax.set_xlim([-0.1, 1.1])
  ax.set_ylim([-0.1, 1.1])
  m, c = -(W[1]/W[2]), -(W[0]/W[2])
  plt.title("parameters : w0 = {}, w1 = {}, w2 = {}".format(W[0], W[1], W[2]))
  ax.plot(X, m * X + c )
  plt.plot()

# define learn function
def learn(W,X,Y):
  visualize(W, X)
  for i in range(Y.size):
    for j in range(W.size):
      W[j] = W[j] + 0.2*(Y[i] - perceptron(W,X[i]))*X[i,j]
    visualize(W, X)

# Parameters
W = numpy.array([0.8, -1, -1]) # w0,w1 and w2 are initialized to 0.8, -1, -1

# learning data x0 x1 x2 y
X = numpy.array([[1, 0, 0], [1, 1, 1], [1, 0, 1], [1, 1, 0]])
Y= numpy.array([0, 1, 0, 0])

#call learn
learn(W,X,Y)
```


parameters : w0 = 0.8, w1 = -1.0, w2 = -1.0


parameters : w0 = 0.6000000000000001, w1 = -1.0, w2 = -1.0


parameters : w0 = 0.8, w1 = -0.8, w2 = -0.8


parameters : w0 = 0.8, w1 = -0.8, w2 = -0.8

2.3.2 AND perceptron with TensorFlow

```
#importing libraries
import numpy as np
import tensorflow as tf
from pylab import ylim, plot
from matplotlib import pyplot as plt

#input set
x = np.array([[1, 0, 0], [1, 1, 1], [1, 0, 1], [1, 1, 0]])

#desired output
y = np.array([0, 1, 0, 0])

#weights
w = np.array([0.8, -1, -1])

#learning rate
n = 0.2

#learning process
print("0 - initial parameters are : w0 = {}, w1 = {}, w2 = {}".format(w[0], w[1], w[2]))
for i in range(len(y)):
        tmp = np.heaviside(tf.experimental.numpy.dot(x[i], w), 0)
        w += tf.subtract(y[i] - tmp , 0) * n * x[i]
        print(i+1,"- new parameters are:w0 = {}, w1 = {}, w2 = {}".format(w[0], w[1], w[2]))
```

```
0 - initial parameters are : w0 = 0.8, w1 = -1.0, w2 = -1.0
1 - new parameters are: w0 = 0.6000000000000001, w1 = -1.0, w2 = -1.0
2 - new parameters are: w0 = 0.8, w1 = -0.8, w2 = -0.8
3 - new parameters are: w0 = 0.8, w1 = -0.8, w2 = -0.8
4 - new parameters are: w0 = 0.8, w1 = -0.8, w2 = -0.8
```

3 CONCLUSION

To sum up, the simple perceptron is a simple neural network that is used to solve linear problems and it contains only one layer consisting of four main parts:

- Input layer
- Weights and bias.
- Input sum.
- Activation function[7]

4 MULTILAYER PERCEPTRON

4.1 PRESENTATION AND ARCHITECTURE

Now that we have defined a perceptron as a neural network with two layers (input layer and output layer), the multilayer perceptron[8] (MLP) is basically a combination of simple perceptrons but with at least one hidden layer.

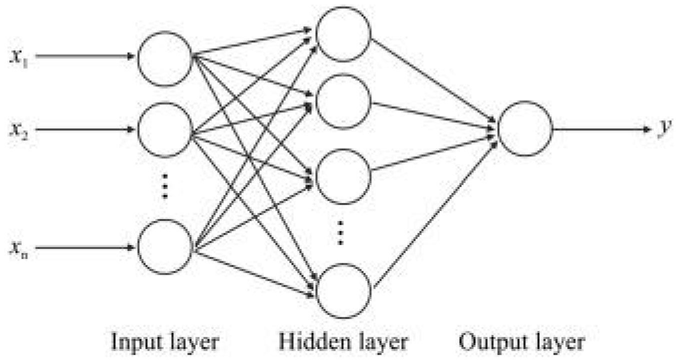


Figure 6 multilayer perceptron

Since MLPs are fully connected, each node in one layer connects with a certain weight ***wij*** to every node in the following layer.

The problem with the previous model (used with the single perceptron) is that a small change in weights or biases can cause a big change in the output. To fix that, instead of just comparing ***∑wj*xj*** to the bias, we will apply the result of z = (∑wj*xj + bias) to a function, called the sigmoid function, noted σ (the result will always be between 0 and 1).

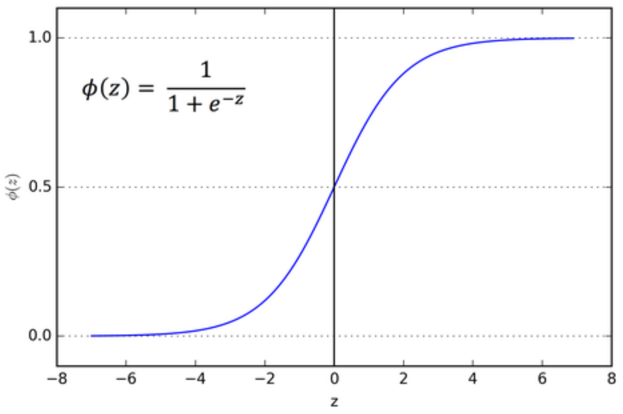


Figure 7 Curve of the sigmoid function

Once configured, the neural network needs to be trained on our dataset (MNIST in our case). Multilayer perceptrons are often applied to supervised learning problems. They train on a set of input-output pairs and learn to model the correlation (or dependencies) between those inputs and outputs. Training involves adjusting the parameters, or the weights and biases, of the model to minimize error. Backpropagation[9] is used to make those weights and bias adjustments relative to the error.

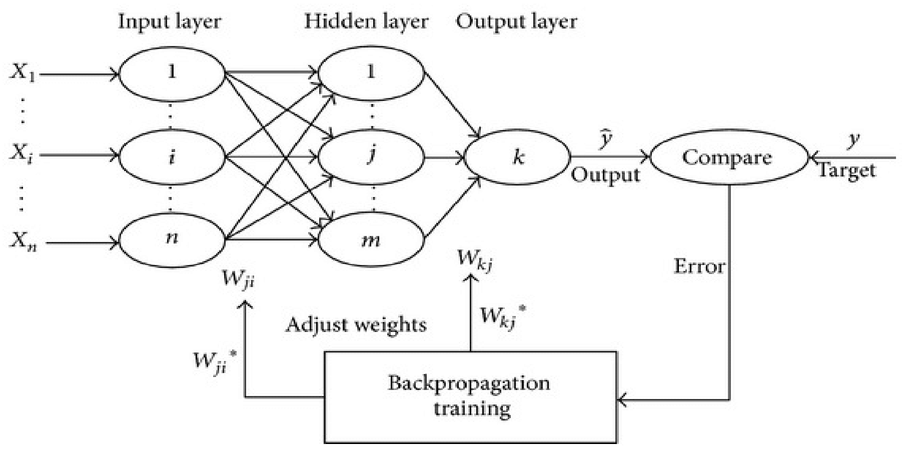


Figure 8 Backpropagation

Backpropagation, short for "backward propagation of errors," is a popular method that calculates the gradient of the error function with respect to the neural network's weights. It is fast, simple, and easy to program. Also, it has no parameters to tune apart from the numbers of input.

Considering the previous network, the backpropagation algorithm is based on the following steps:

- Calculate the output for every neuron from the input layer, to the hidden layer, to the output layer.
- Calculate the error in the outputs (Error = Actual Output – Desired Output).
- Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

We keep repeating this process until the desired output is achieved.

4.2 APPLICATIONS

- **Self-driving cars[10]:**

  In this example, the MLP model consists of two parts; target lane and trajectory models. In order to develop an intuitive and accurate prediction algorithm, a lane-based trajectory prediction model is introduced based on the fact that vehicles drive within a lane except for during lane changes. These target lane and trajectory models enable stochastic MLP modeling and training. The proposed MLP model outputs probabilities of how likely a vehicle will follow each trajectory and each lane for a given input of vehicle position history including current position.

- **Text recognition[11]:**

  After processing the image, each line of the image is identified by an algorithm in which characters are isolated from the image and classified by the basic backpropagation. Now it's time for training our model through a database. The next step is to use a learning set to make sure our model is able to recognize characters.

- **Face Recognition[12]:**

  To recognize a face, the network sets about the task of analyzing the individual pixels of an image presented to it at the input layer. One of the popular uses of face recognition is unlocking phones by taking a picture of your face. The network here analyzes the picture and unlocks the phone if the input picture matches the desired output picture.

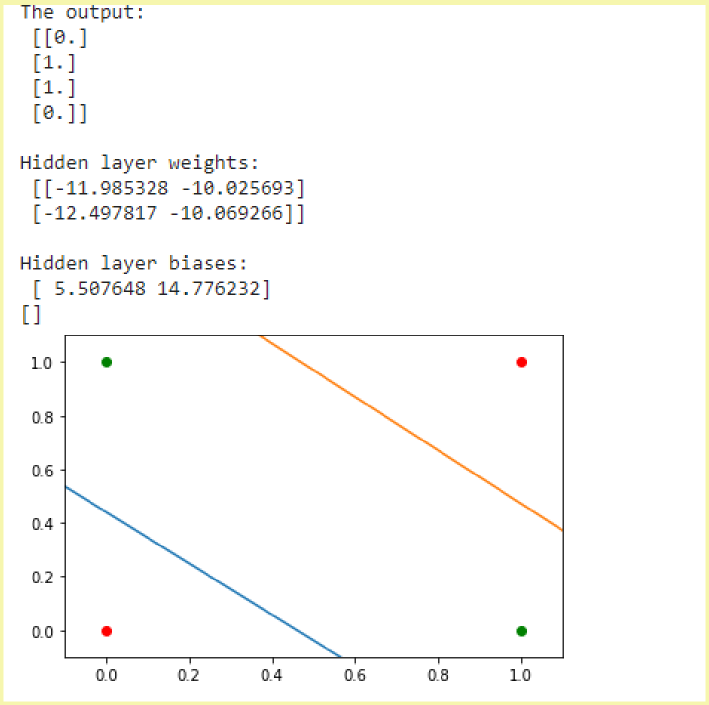
- **XOR:**

  Basically, the utility of Multilayer perceptrons comes after the inefficiency of single perceptrons. A good example is a non-linear problem we have mentioned before (XOR). It can not be solved with single perceptrons. Instead, the solution is to utilize a multilayer perceptron.

4.3 IMPLEMENTATIONS

In this section, we will implement the XOR gate using Keras.

```
#importing libraries
from tensorflow import keras
import numpy as np
from pylab import ylim, plot
from matplotlib import pyplot as plt


#inputs
xor_data = np.array([ [0,0], [0,1], [1,0], [1,1] ])

#outputs
desired_output = np.array([[0],[1],[1],[0]])

#creating the model
model = keras.Sequential()
       #the hidden layer
model.add(keras.layers.Dense(2, activation="sigmoid", input_shape=(2,)))
       #the output layer
model.add(keras.layers.Dense(1, activation="sigmoid"))

#compile the model
model.compile(optimizer=keras.optimizers.Adam(lr=0.2) , loss="binary_crossentropy",
metrics=['accuracy'])
model.fit(xor_data, desired_output, epochs=5000, verbose=0)

#print the output, the weights and biases
predict = model.predict(xor_data)
print("The output: \n", np.round(predict))
arr_weights = np.array(model.layers[0].get_weights()[0])
arr_bias = np.array(model.layers[0].get_weights()[1])
print('\nHidden layer weights: \n',arr_weights)
print('\nHidden layer biases: \n',arr_bias)

#datavisualization
figure, axis = plt.subplots()
X1 = np.arange(-0.2, 1.2, 0.1)
X2 = np.arange(-0.2, 1.2, 0.1)
axis.scatter(0, 0, color="r")
axis.scatter(0, 1, color="g")
axis.scatter(1, 0, color="g")
axis.scatter(1, 1, color="r")
axis.set_xlim([-0.1, 1.1])
axis.set_ylim([-0.1, 1.1])
m1, c1 = -(arr_weights[0][0]/arr_weights[1][0]), -(arr_bias[0]/arr_weights[1][0])
m2, c2 = -(arr_weights[0][1]/arr_weights[1][1]), -(arr_bias[1]/arr_weights[1][1])
plt.plot(X1, m1 * X1 + c1 )
plt.plot(X2, m2 * X2 + c2 )
plt.plot()
```



```
The output:
 [[0.]
 [1.]
 [1.]
 [0.]]

Hidden layer weights:
 [[-11.985328 -10.025693]
 [-12.497817 -10.069266]]

Hidden layer biases:
 [ 5.507648 14.776232]
[]
```

5 CONCLUSION

It is clear that, unlike the simple perceptron, training a multilayer perceptron is often quite slow, requiring thousands or tens of thousands of epochs for complex problems. However, the strength of multilayer perceptron networks lies in that they are theoretically capable of fitting a wide range of nonlinear functions with a high level of accuracy.

CHAPTER 3: CONVOLUTIONAL NEURAL NETWORKS

1 INTRODUCTION

A CNN is a type of neural network that is most applied to image processing problems. It simplifies the image by applying features so it can be better processed. What makes it unique from other neural networks is that unlike the regular neural network (where we connect each neuron to all neurons existing in the previous layer within a special weight), the layers in a CNN are arranged in such a way to detect simpler patterns first and complex patterns further along. In other words, CNN is made up of multiple layers besides the fully connected layer. Those extra layers (the convolutional layer and the pooling layer) are what makes it distinctive. We will explain each one of those in the next paragraphs.

2 PRESENTATION AND ARCHITECTURE

In typical feed-forward neural networks, every neuron in one layer is connected to every other in the next (also known as "**Dense connectivity**"). This leads to a large number of parameters that the network needs to learn. This means we'll need a lot of training data and the time of training will increase as well. On the other hand, CNN can reduce the number of parameters through "**Sparse connectivity**". Sparse connectivity is achieved (inside the convolutional layer) by making the kernel size smaller than the input image which results in a reduced number of connections between layer ***m*** and layer ***m+1***.

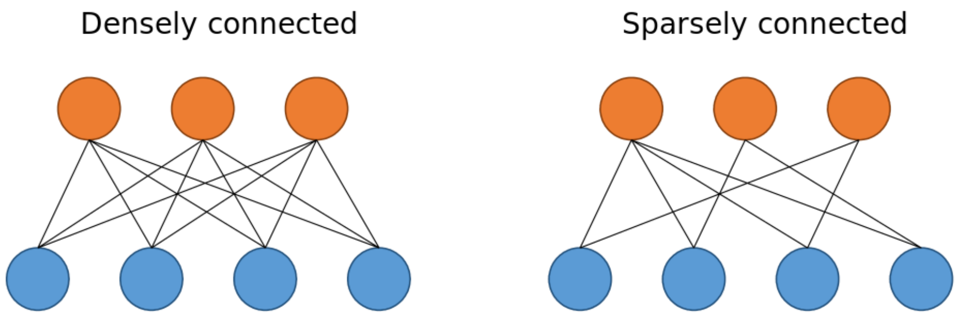


Figure 9 Dense connection vs Sparse connection

In a sparse network, it's more likely that neurons are actually processing meaningful aspects of the problem. For example, in a model detecting cats in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is about a building.

To understand more, it is necessary to take a look at the architecture of a CNN.

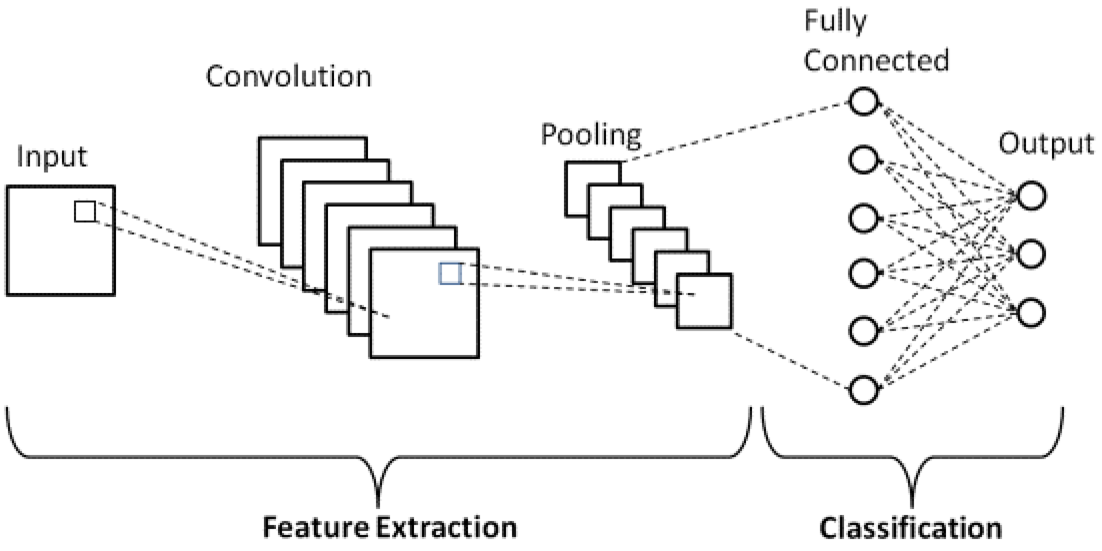


Figure 10 CNN architecture

Apparently, CNN involves two main partitions. The first one consists of “Feature Extraction” and the second one is the “Classification”. To extract features, we use the Convolutional layers and the pooling layers. While we rely on the fully connected layer to classify data.

The convolutional layer:

In the context of a convolutional neural network, a convolution is a linear operation that involves the dot product of an array of input data and an array of weights called a filter (or a kernel). The type of multiplication applied is a dot product.

$$\begin{array}{ccccc} & & \begin{matrix}\vec{b_1} & \vec{b_2}\\ \downarrow & \downarrow\end{matrix} & & \\ \begin{matrix}\vec{a_1}\rightarrow\\ \vec{a_2}\rightarrow\end{matrix} & \underset{A}{\begin{bmatrix}1&7\\2&4\end{bmatrix}} \cdot & \underset{B}{\begin{bmatrix}3&3\\5&2\end{bmatrix}} & = & \underset{C}{\begin{bmatrix}\vec{a_1}\cdot\vec{b_1} & \vec{a_1}\cdot\vec{b_2}\\ \vec{a_2}\cdot\vec{b_1} & \vec{a_2}\cdot\vec{b_2}\end{bmatrix}}\end{array}$$

Figure 11 dot product

In CNN, we may have many convolutional layers. Each layer has its own filters. The filter is applied systematically to each filter-sized path of the input data, left to right, top to bottom. That's why the filter must be smaller than the input data. These filters are applied systematically across an input set (generally an image) to detect specific features in the input so as to be able to discover that feature anywhere in the image.

After multiplying the filter (feature detector) with the input array (the image) multiple times, the result is called a feature map.

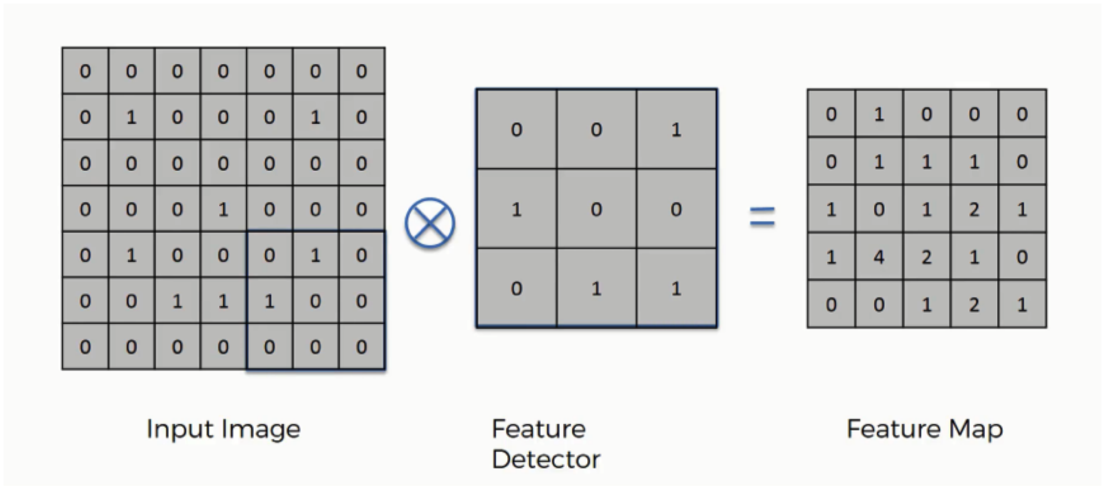


Figure 12 convolutional operation to calculate a feature map

Once we have extracted the feature maps, we can apply a non-linearity (such as the activation function ReLU).

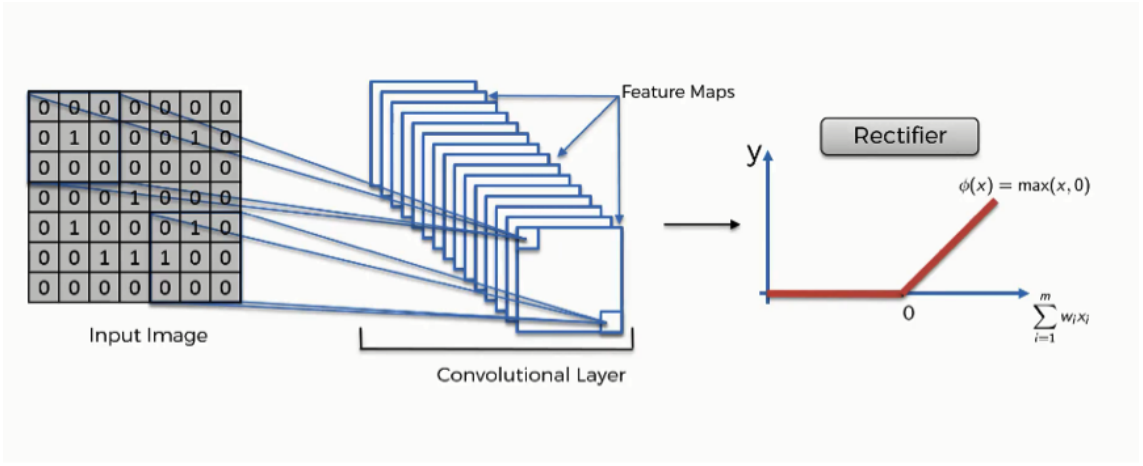


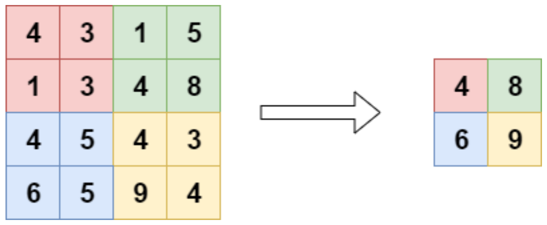
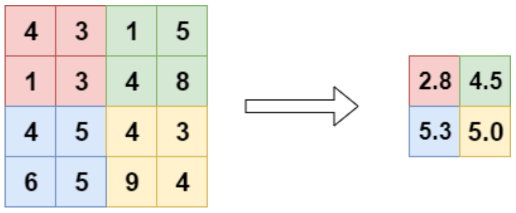
Figure 13 ReLU layer

A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down-sample the feature maps. What we mean by downsampling the feature is making it more robust to changes in the position of the feature in the image. Here where we need a pooling layer.

The pooling layer:

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training the model.

There are two types of Pooling:

Type	Max Pooling	Average Pooling
Objectif	returns the **maximum value** from the portion of the image covered by the Kernel.	returns the **average of all the values** from the portion of the image covered by the Kernel.
Illustration		
Comparison	• Max Pooling also performs as a **Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. • Max Pooling extracts more pronounced features like edges.	• Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. • Average Pooling extracts features more smoothly.

The fully connected layer:

Fully connected layers are often used as the final layers of a CNN. They are simply a typical MLP. In fact, the input for those layers is the output from the previous convolutional/pooling layer. However, before feeding data to the fully connected layers, we need to perform an operation called “flattening data”. And that’s because we have to transform our data into vectors.

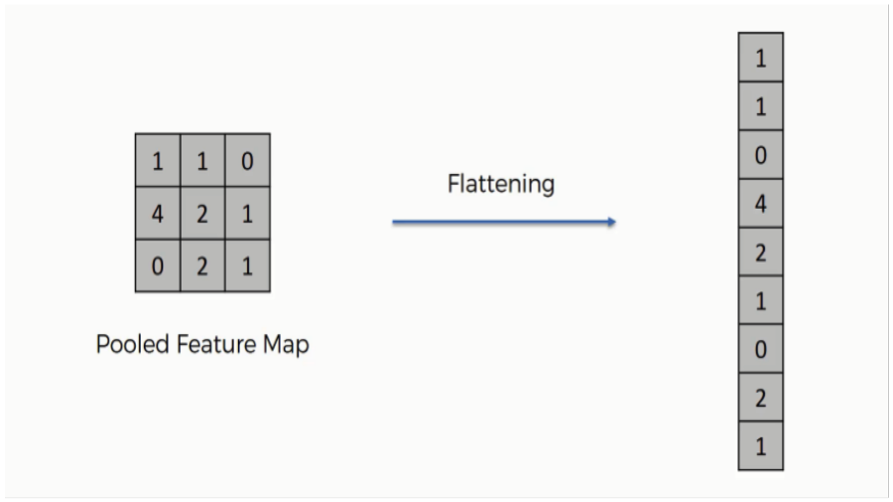


Figure 14 Flattening

The flattened vector is connected to the first layer of the fully connected layers.

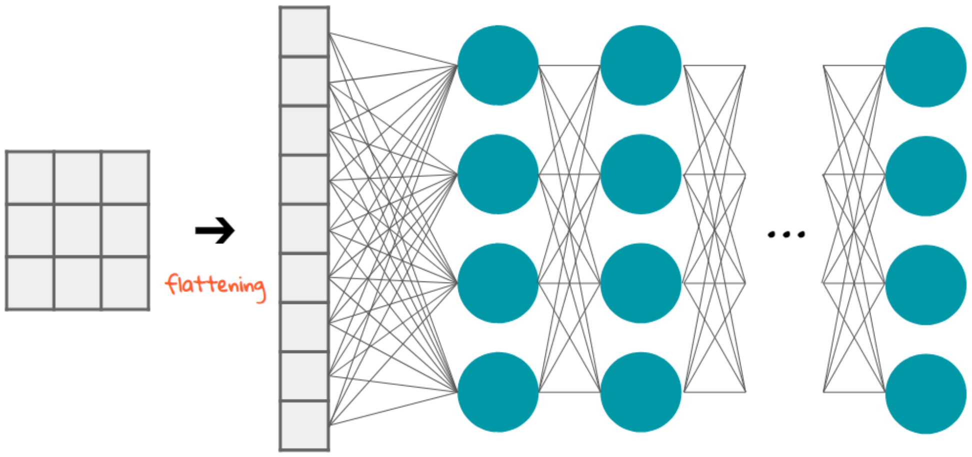


Figure 15 Fully Connected Layer

At this point, we perform for each layer the classic mathematical operations already explained in the previous chapter (adding a bias to the weighted sum and then applying an activation function). As a matter of fact, we usually use ReLU as an activation function instead of using sigmoid.

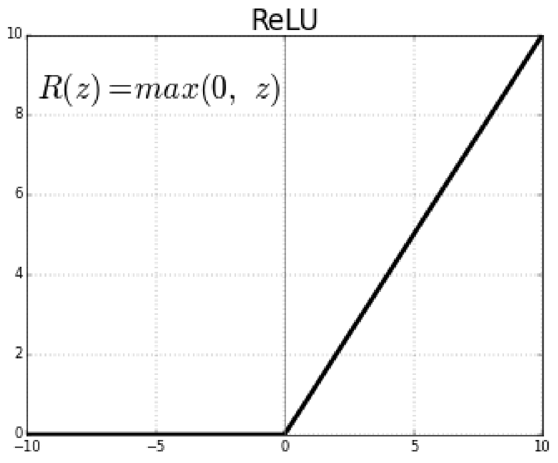


Figure 16 ReLU

The rectified linear activation function (or ReLU for short) is a non-linear activation function that outputs directly the input if it's positive, otherwise, it outputs zero. One advantage of ReLU is that It's sparsely activated (we talked about sparse connectivity in the previous paragraphs). Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all. And this is often desirable (to reduce the number of parameters and training time).

After passing through the fully connected layers, the final layer uses ***the softmax activation function*** instead of ReLU. The softmax function is used for multi-class classification. It transforms the output into values between 0 and 1 so that they can be interpreted as probabilities. Its formula is similar to ***sigmoid.*** In practice, ***Softmax*** is used for multi-classification **in the** Logistic Regression model, whereas ***Sigmoid*** is used for binary classification **in the** Logistic Regression model.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Figure 17 softmax

And so finally, we have the probabilities of the object in the image belonging to the different classes.

3 APPLICATIONS

- CNN on the ImageNet:

ImageNet is a large dataset of annotated photographs intended for computer vision research. The goal of developing the dataset was to provide a resource to improve methods for computer vision. Each image in this database is 224 by 224 pixels.Which is to say, processing a dataset of this size requires a great amount of computing power in terms of CPU, GPU, and RAM. Yet, compared to standard feedforward neural networks with similarly sized layers, CNNs have much fewer connections and parameters and so they are easier to train.

In 2012 Krizhevky, Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton designed a large deep CNN called AlexNet[13] to classify ImageNet data. The architecture of AlexNet is the same as LeNet-5 but much bigger. It is made up of 8 trainable layers. Among them, 5 convolutional layers and 3 fully connected layers.

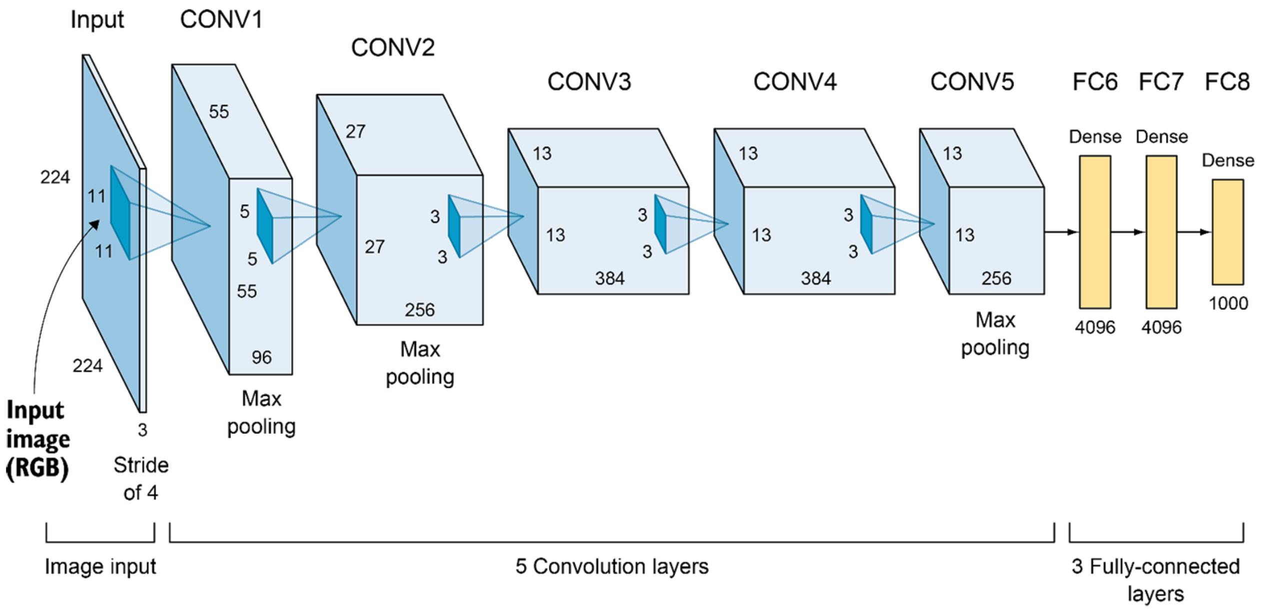


Figure 18 AlexNet architecture for ImageNet classification

- CNN on the image captioning:

Image Captioning is the process of generating a textual description of an image. It uses both Natural Language Processing[14] (NLP) and Computer Vision to generate the captions.

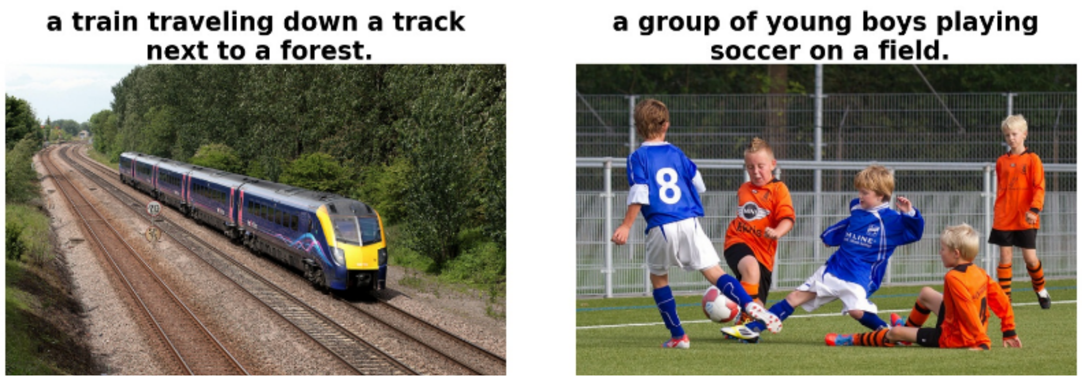

a train traveling down a track next to a forest.
a group of young boys playing soccer on a field.

Figure 19 an example of image captioning

Although it is an easy task for humans to describe an image, it becomes very difficult for a machine to perform such a task. Image captioning does not only require detecting objects in an image but also figuring out how these objects are related to each other. Moreover, semantic knowledge should be expressed in a natural language, which requires a language model to be developed based on visual understanding.

This task follows an encoder-decoder structure where the image is encoded in a dense vector by a CNN followed by a fully connected layer (used to recognize objects). The resulting feature vector is fed to an LSTM[15] to be decoded (used to generate a text sequence, i.e. the caption).

- **CNN on health care:**

Effectively classifying medical images play an essential role in aiding clinical care and treatment. For example, Analysis X-ray is the best approach to diagnose pneumonia which causes about 7593 people to die every year in Morocco (according to the latest statistics) but classifying pneumonia from chest X-rays needs professional radiologists which is a rare and expensive resource for some regions. A logical step to overcome was to create intelligent machines which could learn features needed for image understanding and extract them on their own. One such intelligent and successful model is the CNN model, which automatically learns the needed features and extracts them for medical image understanding.

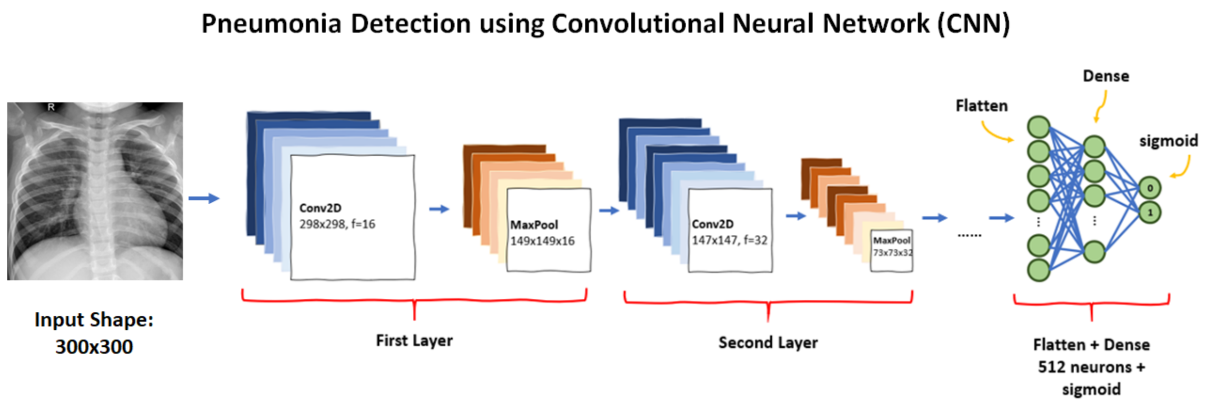


Figure 20 Pneumonia detection using CNN

CHAPTER 4: HANDWRITTEN DIGITS RECOGNITION

1 INTRODUCTION

In the previous chapter, we have explained CNN in a nutshell. It is a variation of a multi-layer perceptron that is popular for analyzing visual imagery.

To accomplish the task of handwritten digit recognition, a model of the convolutional neural network is developed and analyzed for suitable different learning parameters to optimize recognition accuracy and processing time.

2 THE PROPOSED CNN ARCHITECTURE

In this paragraph, we'd like to briefly explain our model's architecture mentioning the hyperparameters of our network. The following figure shows the layers of our model.

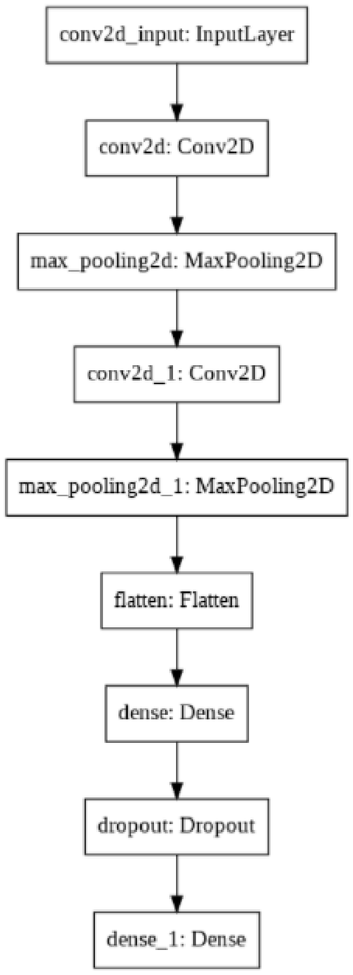


Figure 21 Our model's architecture

During the implementation of our model, we used in the convolutional layers kernels of size 5 and a stride equals 1. We applied 8 filters inside the first convolutional layer and 16 for the second one. Concerning the pooling layers, the strides used are (2,2).

The used kernels are initialized by the Keras initializer defined in code as `tf.keras.initializers.VarianceScaling().`

Jumping to the fully connected layers, the first dense layer contains 128 neurons. The other one contains 10.

The activation functions used are ReLU and softmax (as mentioned in the previous chapter). Our learning rate is equal to 0.001.

We train our model by adopting ***Adam*** as an optimizer[16] and ***categorical cross-entropy*** as a loss function[17]. The reason why we used Adam instead of other optimizers is explained in the figure below:

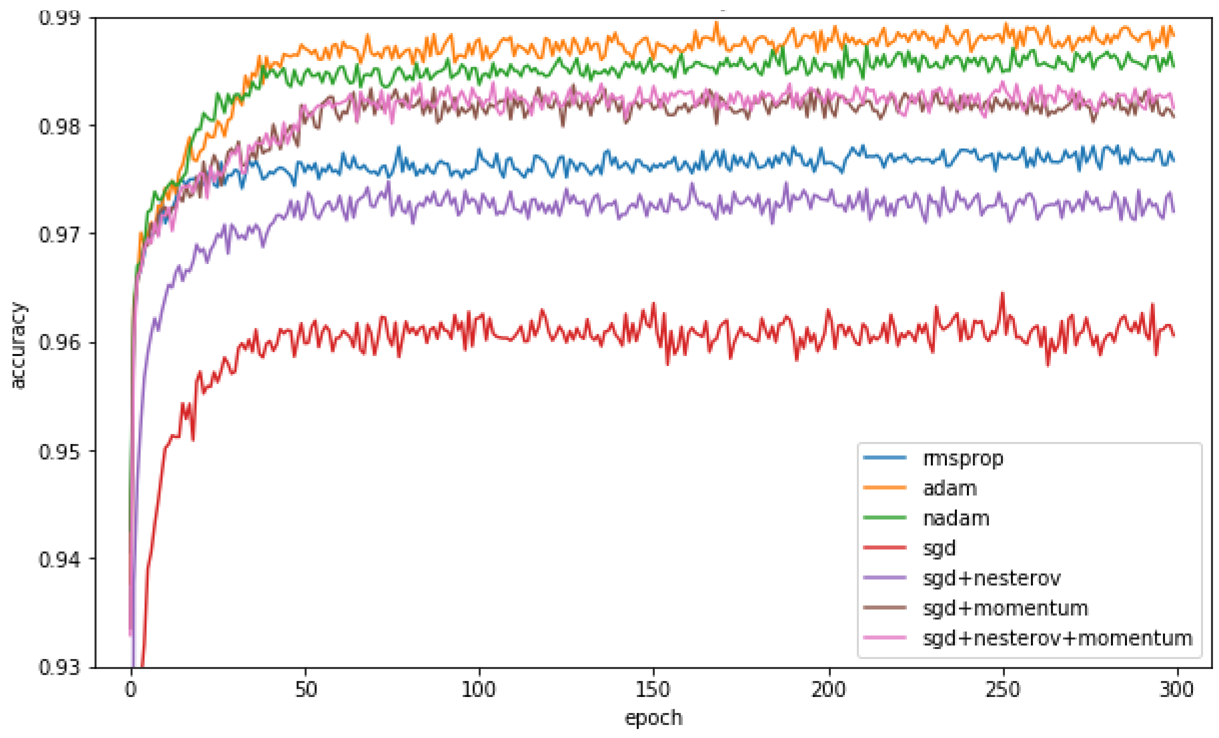


Figure 22 Optimizers comparison

Clearly, Adam takes higher accuracy than other optimizers in terms of epochs. As for the loss function used (categorical cross-entropy), it calculates the loss of an example by computing the following sum:

$$Loss = -\Sigma y \, . \, log(y')$$

Where y' is the value of output and y is the corresponding target value. In this context, we have images of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The model uses the categorical cross-entropy to learn to give a high probability to the correct digit and a low probability to the other digits.

Now it should be clear why we use the softmax activation function with categorical cross-entropy. Considering an array of calculated outputs, each value of this array is going to be inside the base-2 logarithm.

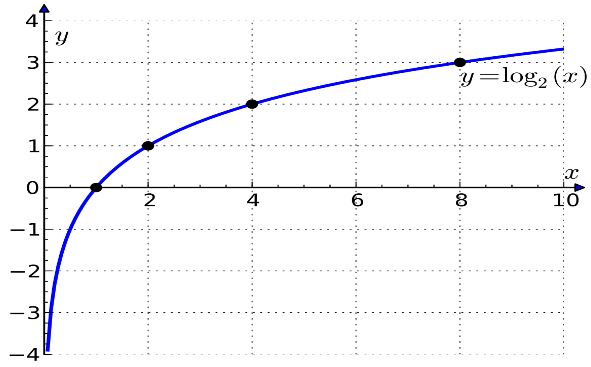


Figure 23 Base 2 logarithm

Knowing that we can't pass values in rang]-00,0] inside log_2. We need to make sure that the calculated outputs respect the definition domain of log_2. The only way to get those values is by using the softmax activation function.

Finally, due to the previous parameters, only 5 epochs were capable of training our model with an accuracy of 0.98.

3 EXPERIMENTS AND RESULT

3.1 THE USED DATASET

MNIST(Modified National Institute of Standards and Technology) database contains handwritten digits. It is a subset of the larger dataset present in NIST(National Institute of Standards and Technology). Developed by Yann LeCunn, Corinna Cortes and Christopher J.C. Burges and released in 1999. This is a "hello world" dataset deep learning in computer vision beginners for classification, containing ten classes from 0 to 9. The original black and white images of NIST had been converted to grayscale in dimensions of 28*28 pixels in width and height, making a total of 784 pixels. Pixel values range from 0 to 255, where higher numbers indicate darkness and lower as lightness.

MNIST database consists of two NIST databases – Special Database 1 and Special Database 3. Special Database 1 contains digits written by high school students. Special Database 3 consists of digits written by employees of the United States Census Bureau.

The MNIST dataset contains 70,000 images of handwritten digits (zero to nine) that have been size-normalized and centered in a square grid of pixels. Each image is a 28 × 28 × 1 array of floating-point numbers representing grayscale intensities ranging from 0 (black) to 1 (white). The target data consists of one-hot binary vectors of size 10, corresponding to the digit classification categories zero through nine.

The highest error rate is 12%. The original paper of MNIST showed the report of using SVM(Support Vector Machine) gave an error rate of 0.8%.

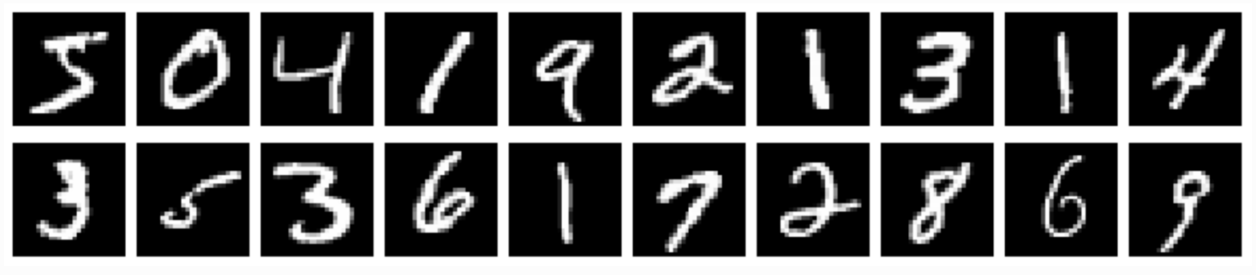


Figure 24 Example of MNIST images.

To load the MNIST dataset , we use the following code :

```
mnist_dataset = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist_dataset.load_data()
```

The MNIST dataset will be loaded as a set of training and test inputs (X) and outputs (Y). The inputs are samples of digit images while the outputs contain the numerical value each input represents. The exact shape of each dataset is provided below for your convenience.

```
print('MNIST Dataset Shape:')
print('x_train: ' + str(x_train.shape))
print('y_train: ' + str(y_train.shape))
print('x_test: ' + str(x_test.shape))
print('y_test: ' + str(y_test.shape))
```

```
MNIST Dataset Shape:
x_train: (60000, 28, 28)
y_train: (60000,)
x_test: (10000, 28, 28)
y_test: (10000,)
```

3.2 THE IMPLEMENTATION OF OUR MODEL

To solve our main problem(handwritten digits recognition), we decided to implement a Convolutional Neural Network model and train it with the MNIST dataset. The CNN model will be able to 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.

After **importing the right libraries** and dependencies we should start by **loading the MNIST data** and **preparing** it for the model:

```
mnist_dataset = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist_dataset.load_data()
```

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

```
x_train = x_train / 255
x_test = x_test / 255
```

```
n_classes = 10
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
```

The second step is to **build the model** by defining its layers:

```
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Convolution2D(
    input_shape=(28,28,1),
    kernel_size=5,
    filters=8,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.add(tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=(2, 2)
))

model.add(tf.keras.layers.Convolution2D(
    kernel_size=5,
    filters=16,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.add(tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=(2, 2)
))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(
    units=128,
    activation=tf.keras.activations.relu
));

model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(
    units=10,
    activation=tf.keras.activations.softmax,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))
```

Before jumping to the last step, We'd like to give a simple explanation to the dropout layer since we haven't mentioned it before. Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. And that is why we said CNN is sparsely connected, since some neurons aren't activated during training. The reason why we do this is to reduce the model capacity so that our model can achieve lower generalization error.

Finally, we have to **compile**, **train**, **save** and then **evaluate** our model:

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

```
history = model.fit(x_train, Y_train,
          batch_size=128, epochs=5,
          verbose=2,
          validation_data=(x_test, Y_test))
```

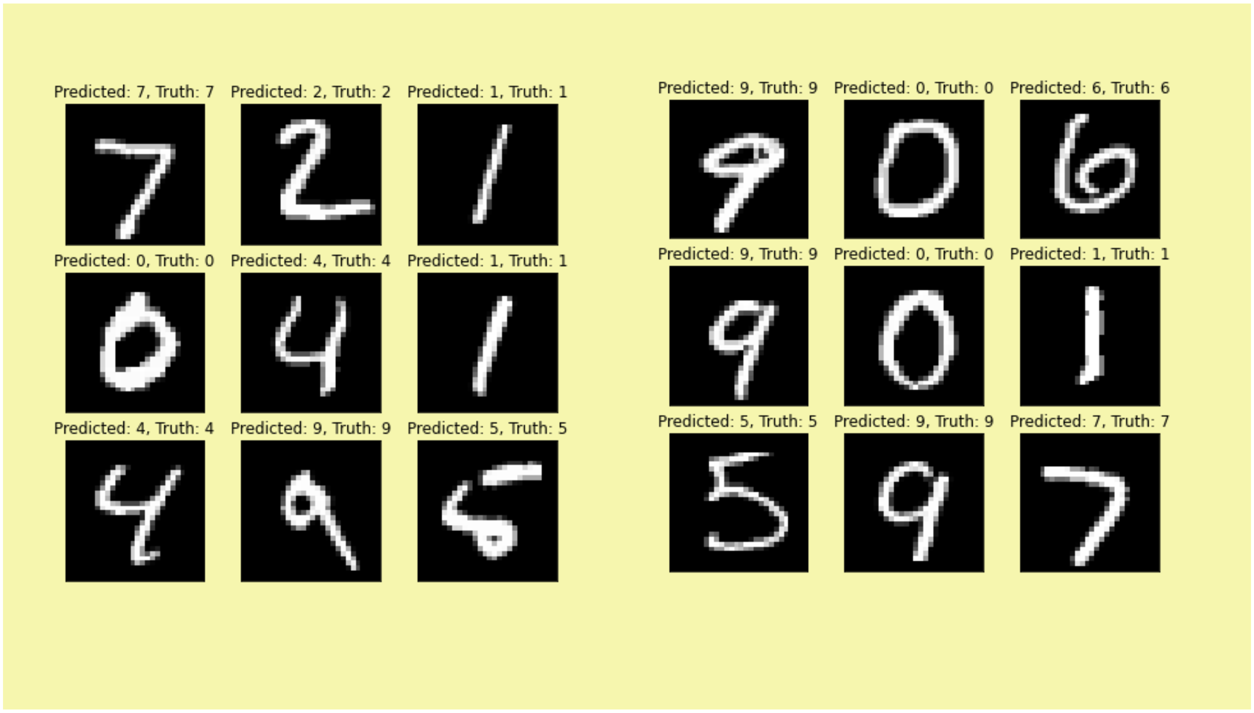
```
model.save('mnist.h5')
```

```
mnist_model = load_model('mnist.h5')
loss_and_metrics = mnist_model.evaluate(x_test, Y_test, verbose=2)

print("Test Loss", loss_and_metrics[0])
print("Test Accuracy", loss_and_metrics[1])
```

```
313/313 - 2s - loss: 0.0342 - accuracy: 0.9877
Test Loss 0.03416096046566963
Test Accuracy 0.9876999855041504
```

With an accuracy that can reach 0.9877, the model could recognize almost 99% of the images fed into it. The following code visualizes some of the images that could be well-classified by our model:

```
#data visualisation
predicted_classes = mnist_model.predict_classes(x_test)

correct_indices = np.nonzero(predicted_classes == y_test)[0]
print()
print(len(correct_indices)," classified correctly")

plt.rcParams['figure.figsize'] = (7,14)
figure_evaluation = plt.figure()

# plot 18 correct predictions
for i, correct in enumerate(correct_indices[:18]):
    plt.subplot(6,3,i+1)
    plt.imshow(x_test[correct].reshape(28,28), cmap='gray', interpolation='none')
    plt.title(
      "Predicted: {}, Truth: {}".format(predicted_classes[correct],
                                        y_test[correct]))
    plt.xticks([])
    plt.yticks([])

figure_evaluation
```

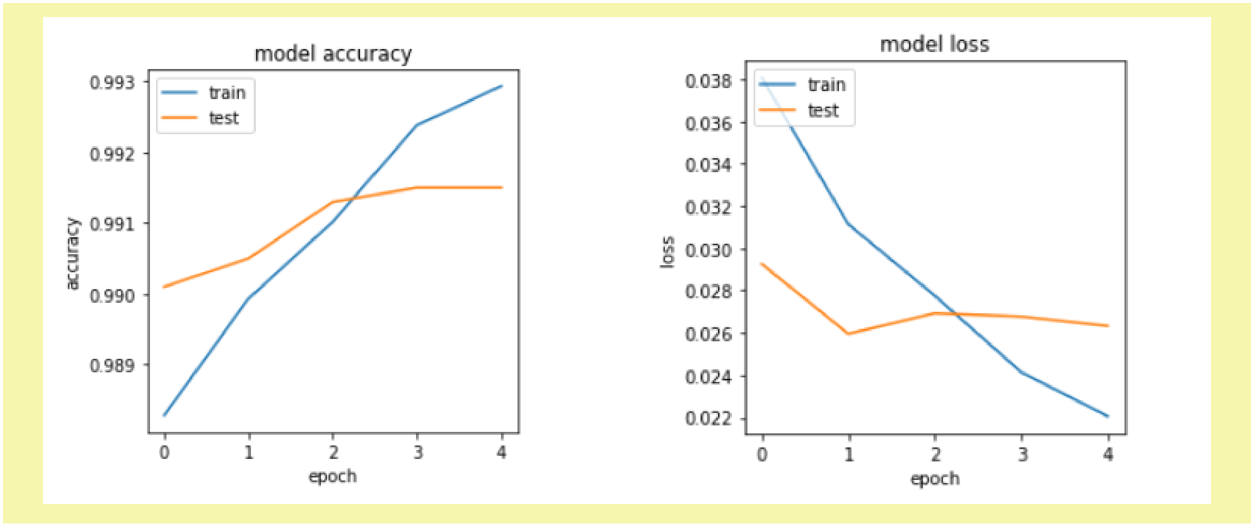


3.3 EVALUATIONS AND RESULTS

Above points, we described our model's architecture and hyperparameters. Here, we are executing the code below to plot the loss and the accuracy of our model throughout training and testing.

```
# summarize history for accuracy
plt.figure(figsize=(4, 4))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.figure(figsize=(4, 4))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

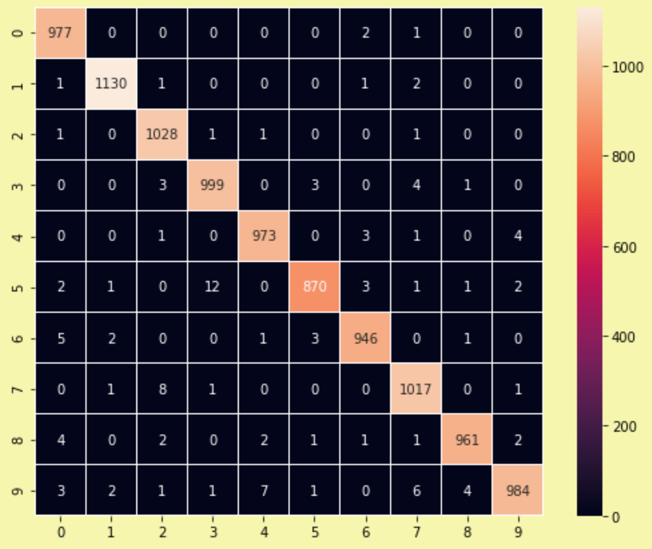
The result is the graphs below:



The reason it is useful to visualize the accuracy and the loss is that it helps us make informed decisions about the architectural choices that need to be made. In simple words, we would know if the hyperparameters we choose are the right ones or they need to be updated.

For the same reasons we are going to plot a confusion matrix that is going to give us a better idea of what our classification model is getting right and what types of errors it is making and help us know exactly what numbers that the model thinks look similar. Using this matrix we can clearly see that images of “1” are the most correctly classified by our model and images of “5” are the least.

```
predictionss = model.predict([x_test])
predictions = np.argmax(predictionss, axis=1)
confusion_matrix = tf.math.confusion_matrix(y_test, predictions)
f, ax = plt.subplots(figsize=(9, 7))
sn.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.5,
    fmt="d",
    square=True,
    ax=ax
)
plt.show()
```



	0	1	2	3	4	5	6	7	8	9
0	977	0	0	0	0	0	2	1	0	0
1	1	1130	1	0	0	0	1	2	0	0
2	1	0	1028	1	1	0	0	1	0	0
3	0	0	3	999	0	3	0	4	1	0
4	0	0	1	0	973	0	3	1	0	4
5	2	1	0	12	0	870	3	1	1	2
6	5	2	0	0	1	3	946	0	1	0
7	0	1	8	1	0	0	0	1017	0	1
8	4	0	2	0	2	1	1	1	961	2
9	3	2	1	1	7	1	0	6	4	984

Straight off, we are going to analyse and discuss our model via a few scenarios. We choose for each scenario different hyperparameters to our model.

- Scenario 1:
  Epochs: 5, **Optimizer**: SGD.
- Scenario 2:
  Epochs: 9, **Optimizer**: SGD.
- Scenario 3:
  Epochs: 5, **Optimizer**: Adam.
- Scenario 4:
  Epochs: 9, **Optimizer**: Adam.

We execute our model for each scenario separately and we write down the results.

Scenario	Accuracy	Loss	Training time
1	0.9638	0.1278	1min 40sec
2	0.9738	0.0836	2min 46sec
3	**0.9895**	**0.0342**	**1min 33sec**
4	0.9896	0.0328	2min 57sec

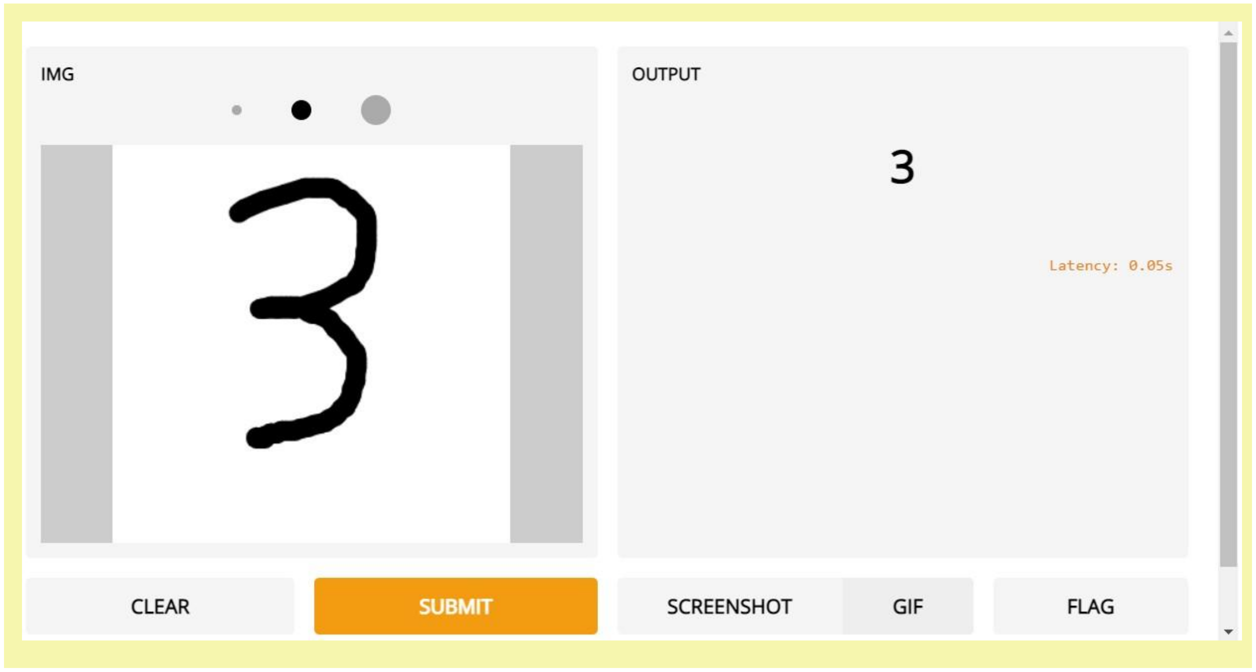
Observing the results above, we reach the best accuracy and the minimum loss in scenario 4. However the training time for this scenario is much larger than other scenarios. Moreover, The accuracy of scenario 3 is very close to the fourth's scenario. The difference is only 0.001 while it only needs one minute and 33 seconds to be trained. For this reason we relied on the third scenario to implement our model.

3.4 USING THE MODEL (GRAPHICAL USER INTERFACE)

After having implemented our model and tested it, it is ready to be used for classifying new images of handwritten digits. For that reason, we need to implement some kind of interface that the user can use to input the digits.

We used a python library called gradio to implement the following GUI code:

```
!pip install -q gradio
import gradio as gr

mnist_model = load_model('mnist.h5')

test=x_test[0].reshape(-1,28,28,1)
pred=mnist_model.predict(test)
print(pred)

#prepare the images
def predict_image(img):
  img_3d=img.reshape(-1,28,28,1)
  im_resize=img_3d/255.0
  prediction=mnist_model.predict(im_resize)
  pred=np.argmax(prediction)
  return pred

#creating the GUI
iface = gr.Interface(predict_image, inputs="sketchpad", outputs="label")

#launching the GUI
iface.launch(debug='True')
```



4 CONCLUSION

The last chapter was devoted to the main problem of our project. We could choose a CNN architecture for our model, load the MNIST dataset, build the model, evaluate it and visualize results and finally build a GUI to make use of the model.

CONCLUSION

All in all, we were able to successfully program a handwritten digit recognizer using a deep convolutional neural network. The program was created using python libraries for machine learning and was trained with the MNIST dataset for handwritten digits. The program also contains an interactive part that gives the user the ability to input a digit for the program to be recognized using the model.

The working process of this project has helped us explore machine learning step by step and learn new concepts related to deep learning and neural networks. Moreover, it made us more familiar with python and the libraries and frameworks used in the AI field. Without forgetting the technical terms and the different architectures that we have learned.

After testing our model with the MNIST testing data, it has successfully scored an accuracy greater than 98%. However, we think the model can still be improved by considering better hyperparameters and using more complex model architectures.

Finally, we have successfully met our planned goals for this project in the given deadlines and we are satisfied with the results, nevertheless, we will continue to add more functionalities to our program, such as making the GUI more user-friendly and loading the created model to the web.

REFERENCES

[1]: to get more ideas about this field, check the following link:
https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-eed6024fdb08

[2]: Check the original paper of CNN published by Yann LeCun, Patrick Haffner, léon Bottou, and Yoshua Bengio:
http://yann.lecun.com/exdb/publis/pdf/lecun-99.pdf

[3]: This also the original paper of LeNet5 architecture published by Yann LeCun, Patrick Haffner, léon Bottou, and Yoshua Bengio:
http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

[4]: Documentation of Keras:
https://keras.io/

[5]: This paper was published by Haochan wang and Bhiksha Raj about NN:
https://arxiv.org/pdf/1702.07800.pdf

[6]: You can get general ideas about Rosenblatt's perceptron (the first modern NN) here:
https://www.neuroelectrics.com/blog/2016/08/02/artificial-neural-networks-the-rosenblatt-perceptron/

[7]: the article below explains -briefly- the most famous activation function used for deep learning:
https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/#:~:text=for%20Output%20Layers-,Activation%20Functions,a%20layer%20of%20the%20network.

[8]: A useful course on MLP:
https://machinelearningmastery.com/neural-networks-crash-course/

[9]: A paper published by Yann LeCun about the theory of backpropagation:
http://yann.lecun.com/exdb/publis/pdf/lecun-88.pdf

[10]: This tutorial explains the process of training a self-driving car:
https://towardsdatascience.com/how-to-train-your-self-driving-car-using-deep-learning-ce8ff76119cb

[11]: this paper was published by Vijendra Singh, Hem Jyotsna Parashar, and Nisha Vasudeva about recognizing texts using MLP:
https://arxiv.org/ftp/arxiv/papers/1612/1612.00625.pdf

[12]: another paper published on face recognition:
https://liris.cnrs.fr/Documents/Liris-6963.pdf

[13]: AlexNet is a CNN architecture found by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton used mostly in ImageNet dataset classification. The original paper :
https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[14]: A gentle introduction to NLP:
https://builtin.com/data-science/introduction-nlp

[15]: LSTM is out of our topic, however, this should be a very well organized guide to this recurrent neural network:
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

[16]: Various optimization deep learning algorithms:
https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6#:~:text=Optimizers%20are%20algorithms%20or%20methods,order%20to%20reduce%20the%20losses.&text=Optimization%20algorithms%20or%20strategies%20are,the%20most%20accurate%20results%20possible.

[17]: Loss functions:
https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/
https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Some other resources that we used during the entire project:

http://neuralnetworksanddeeplearning.com/

https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

https://www.youtube.com/watch?v=5tvmMX8r_OM&list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI

https://www.youtube.com/watch?v=aiDv1NPdXvU&list=PLWi7UcbOD_0vc_rSRgHAyKig2eMdMTRyo

https://deepai.org/machine-learning-glossary-and-terms/sigmoidal-nonlinearity

https://medium.com/pursuitnotes/day-12-kernel-svm-non-linear-svm-5fdefe77836c

https://medium.com/analytics-vidhya/getting-started-with-python-anaconda-google-colab-and-virtual-environments-1ce8fc3286f9

https://towardsdatascience.com/perceptrons-logical-functions-and-the-xor-problem-37ca5025790a

https://www.allaboutcircuits.com/technical-articles/how-to-train-a-basic-perceptron-neural-network/

https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

https://pub.towardsai.net/the-architecture-implementation-of-lenet-5-eef03a68d1f7

https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9

The code used in this document is available here.