UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

TREBALL DE FI DE GRAU

TFG TITLE: Machine learning with deep neural networks and object tracking applied to motion of airplanes

DEGREE: Grau en Enginyeria d'Aeronavegació

AUTHOR: Claudia Martin Torres

ADVISOR: Pietro Alberto Massignan

DATE: September 4, 2020

Título: Aprendizaje automático con redes neuronales profundas y seguimiento de objetos aplicado al movimiento de aviones
Autor: Claudia Martin Torres

Director: Pietro Alberto Massignan

Fecha: 4 de setembre de 2020

Resumen

El objetivo de este proyecto es comprender los conceptos que subyacen al aprendizaje automático y cómo implementarlos. Para lograr este propósito, se ha realizado un estudio exhaustivo de los orígenes de esta tecnología, describiendo los tipos de redes neuronales más populares, su historia y las arquitecturas e implementaciones correspondientes.
Se presentan tres implementaciones de redes neuronales, utilizando conjuntos de datos conocidos mundialmente. En la última implementación, se ha realizado un estudio exhaustivo para lograr el mejor algoritmo en rendimiento, teniendo en cuenta diferentes configuraciones. En la segunda parte del proyecto se ha utilizado Detectron2, un programa avanzado de aprendizaje automático que realiza detección de objetos. Hemos trabajado con él, ejecutando un estudio del movimiento de aviones en movimiento, implementando un nuevo método para realizar el seguimiento de objetos dado un conjunto de imágenes extraídas de un video.

Title : Machine learning with deep neural networks and object tracking applied to motion of airplanes

Author: Claudia Martin Torres

Advisor: Pietro Alberto Massignan

Date: September 4, 2020

Overview

The aim of this project is to understand the concepts underlying machine learning and how to implement those. To achieve this purpose, an exhaustive study of the origins of this technology has been made, describing the most popular types of neural networks, their history, and the architectures and subsequent implementations.
Three implementations of neural networks are presented, using world-known datasets. In the last implementation, an exhaustive study has been realized to achieve the best performance algorithm taking into account different settings. In the second part of the project, Detectron2 has been used, an advanced machine learning program that performs object detection. We have worked with this program and executed a study of the motion of moving airplanes, implementing a new method to track objects given a set of images extracted from a given video.

CONTENTS

LIST OF FIGURES

INTRODUCTION

Nowadays new technologies are providing extremely fast changes in all areas of knowledge. A Digital Revolution is happening, producing a great impact on society and the way things are done. Machine learning, artificial intelligence, and deep learning are techniques that are changing the world as we understand it.

Arthur Samuel was the first to define machine learning in 1959 as a "Field of study that gives computers the ability to learn without being explicitly programmed". The field had a relatively slow start because computers and devices were not prepared to deal with a large amount of information. However, in the last fifteen years, the advent of the internet and the computing power of modern devices increased enormously the importance of machine learning techniques.

Investment in machine learning and artificial intelligence has grown up so fast during the last years, and the sector forecasts are better than ever. Reliable sources predict that the ML market will grow from 7 billion to 30 billion in the next four years, attaining a CAGR (Compound Annual Growth) of 43%.

Machine learning's growing adoption in business yields algorithms that are more effective as time passes and very complex problems can now be solved extremely quickly. It is a key tool for many companies these days, and more and more job opportunities require it as a skill.

Today machine learning is present in almost everything, for instance in the automotive sector providing driver safety systems, in financial services, healthcare, retail, industrial sector, and helping construct smart buildings. It can be applied to smart robots, video recognition, natural language processing, computer vision platforms, pricing, and an infinity of applications. Nevertheless, these new technologies are still highly unknown, especially the limits that they may have.

This project is devoted to study machine learning, and more specifically deep learning. In particular, we will start by reviewing the most important neural networks, their operation, and their characteristics. We have implemented three neural networks algorithms analyzing two different basic popular datasets in machine learning, the MNIST database, and the CIFAR-10 database. In the last implementation, an exhaustive study has been made, looking for the best algorithm performance comparing various settings.

In the second part of the project, we have worked with an advanced software for image detection, Detectron2, which we have adapted to perform object tracking. We have analyzed a two-dimensional video of some airplanes moving, describing their trajectory and realizing a motion study.

The purpose of the project has been understanding this new whole world of technology, describing the most popular neural networks that nowadays are implemented, and the large number of applications that are present in our life. Moreover, approaching a basic implementation of machine learning algorithms and how they work, and dealing with a more complex program that can be adopted by big companies for different purposes.

Chapter 1 explains the beginning of machine learning and the field of deep learning, Chapter 2 is centered on understanding Deep Neural Networks, and Chapter 3 on a popular Convolutional Neural Network. Finally, Chapter 4 introduces Detectron2 and the practical

part dealing with the implementation of object tracking.

CHAPTER 1. GENERAL INTRODUCTION TO NEURAL NETWORKS AND MACHINE LEARNING

In the first chapter of this project, an introduction to neural networks is made. Biological neural networks are described and the machine learning field is introduced, giving an overview of the basic concepts, such as the history of machine learning and the existing types of implemented algorithms nowadays.

1.1. Neural Networks

First of all it is needed to have a general vision of what a neural network is. An appropriate definition is given at Ref. [1]:

> *A neural network is an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the interunit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns.*

To understand that, it is important to take a quick look into the very first meaning of what a neural network is, the biological one.

Humans have a brain that contains approximately 100 billion nerve cells. A nerve cell (also called neuron) is the fundamental unit of the brain and also of the nervous system. It is a special type of cell whose purpose is to both transfer information around the body and to give orders to it.

Neurons connect other cells via electrical signals or *spikes* with a very short living time. The interneuron connections are made by electrochemical junctions called *synapses*. These are the main component of the nervous tissue in nearly all animals (excluding sponges and placozoa), giving a typical brain 100-1,000 trillion of them.

It is important to understand the structure of a neuron. A neuron consists of a cell body (soma), axons and dendrites:

- *Cell body*: the neuron's core. It carries the genetic information and it contains the nucleus and other specialized organelles. Thus, most protein synthesis happens there and it provides energy to drive activities. The soma is protected by a membrane that can be communicated with its immediate surroundings.
- *Axons*: are finer, long, tail-like structures which carry nerve signals from the soma and give information back to it. They contain a substance called myelin that helps to conduct electrical signals. The soma and the axon are joined by a junction called axon hillock, the most easily excited part of the neuron and the part of the axon where the spike initiates. Neurons generally have one axon, but it is possible to have more (they are not equal between them).

- *Dendrites*: are the receiving part of the neuron. Dendrites are fibrous roots that emerge out from the soma with many branches. They pick up and process the signals from the axons.

Therefore, we can conclude that a neuron is an information processor. The entry channel of this processor are the dendrites, the soma is the processor itself and the exit channel would be the axon or axons (Figure 1.1).

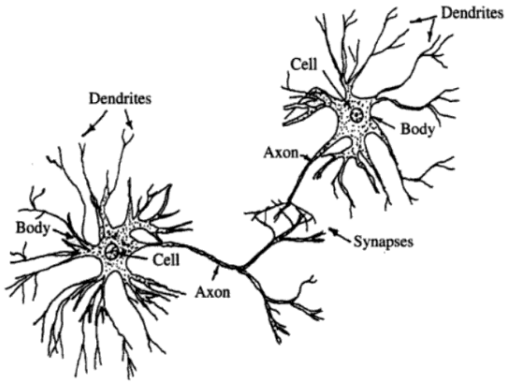


Figure 1.1: Structure of a neuron (Ref. [2])

So, how does a neural network work? As said before, the connections between these nerve cells are the synapses. The synapses are unidirectional connections, in which transmission of information is made electrically at the inner part of the neuron and chemically between neurons; due to specific substances called neurotransmitters. The use of the synapses is possible because of the capability of a neuron to transfer an *action potential*.

To explain action potentials it is necessary to take into account the concentration gradient concept. A concentration gradient is a difference in ion concentrations between the inside and the outside of the neuron, and this causes the action potentials. An action potential is a temporary shift in the neuron's membrane potential caused by these ions flowing in and out of the neuron. During an action potential, part of the neural membrane opens to allow the positively charged ions inside the neuron and allow negatively charged ions out.

So first of all, an impulse is sent out from the cell body. Then, when this impulse reaches a specific threshold, the action potential is fired (that means executed) sending the electrical signal down the axons and concluding with the transmission. This is how neurons act and how they transmit the information between them.

The brain neural network is known to possess 10^{10} to 10^{11} neurons connected to each other, which determines a total number of connections in the network of 10^{13} to 10^{15}. This makes a huge network, and also a difficult structure with complex behavior.

Modeling how a neural network works (or modelling brain function) is an active area of neuroscience research, and nowadays it is still not well understood. Nevertheless, the actual knowledge of this theme has developed the Artificial Neural Networks (ANN) from the machine learning field. This ANN provides a way to approach the performance of the biological neural networks.

1.2. Machine Learning

Machine learning is a subfield of computer science which includes pattern recognition and computational learning theory from artificial intelligence (AI). It is an analysis technique that explores the construction and study of algorithms that can learn (a process that is natural for people and animals) from and make predictions on data. These algorithms learn from the experience of this data, avoiding strictly static program instructions and model equations. They improve adaptively as the number of available sample data rises.

Machine learning is related with computational statistics and mathematical optimization also. With the rise of big data, machine learning has become a key tool to solve problems in areas such as computational finance, computational biology, energy production, image processing and artificial vision, automotive, aerospace and manufacturing, natural language processing, and others. Machine learning is used for tasks that are not possible with programming explicit algorithms, for instance spam filtering, search engines, and computer vision. When used in industrial contexts, machine learning is referred to as predictive modeling or predictive analytics.

Machine learning (1959) is not a new concept. So, why now?

- Data availability: Billions of people are online using connected devices or sensors that generate a large amount of data, that is available to use for instance as training data for learning algorithms, and perform complex tasks.
- Computing power: Nowadays there are powerful computers and the possibility of connecting remote processing power through the Internet makes possible techniques of machine learning that process big amounts of data.
- Innovation of algorithms: New machine learning techniques. Layered neural networks (or "deep learning") are giving new opportunities and there is also a lot of research on this field.

1.2.1. History

The field of artificial intelligence dates back to the 1950s. An IBM researcher, Arthur Lee Samuels, developed a self-learning program for playing checkers (one of the earliest machine learning programs). Indeed, later Samuel defined the term "machine learning" on a paper published in the *IBM Journal of Research and Development:*

> *"Field of study that gives computers the ability to learn without being explicitly programmed"*- Arthur Lee Samuels, 1959 (Ref. [24]).

Pioneering machine learning research was made using simple algorithms by the 1950s. Later, in 1965 Nils J. Nilsson published the book *Learning Machines* of machine learning research emphasizing pattern classification. Also, other books referring to this relation with pattern classification were published during the 1970s, for example the *Pattern Recognition and Scene Analysis* by Duda and Hart. Then, in 1981 Stevo Bozinovsky wrote the *Teaching Space: A Representation Concept for Adaptative Pattern Classification*, that related how to make a neural network learn from a computer terminal.

Furthermore, it is important to point out Tom M. Mitchell's definition about the setting of machine learning (1997): "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". Moreover, Alan Turing's proposal in his paper also was crucial *Computing Machinery and Intelligence* in 1950, where he questioned the capability of a machine to act as a thinking entity.

During the 1990s, programs for computers to analyze and manage large amounts of data and learning were developed. The 2000s introduced new machine learning methods such as support vector clustering and Kernel methods (algorithms for pattern analysis); and unsupervised machine learning methods became more used. Nowadays, deep learning it is widely used for software and developing of applications, being a very interesting field of research.

1.2.2. Types of machine learning algorithms

The types of machine learning algorithms differ in the nature of the problem to be solved, the type of data (inputs and outputs), and the volume of the data. They can be classified into four broad categories: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

- Supervised learning: It typically begins with an established set of data that contains the inputs and the desired outputs. This is called the *training data*, which is labeled by the so-called *supervisor* (they usually are humans, but they can also be machines) explaining the meaning of the data. The supervised learning algorithm is expected to find patterns in data that can be applied to an analytic procedure, and thus recognize and understand how the data is classified by the user (learn).

  Apart from the training data, it also exists the *test data*, that can be defined as another subset of data that evaluates the performance of an algorithm once it has been trained, and it is useful to know important parameters of an algorithm; for instance the accuracy and loss, that will be introduced later.

  There are two types of supervised learning algorithms:

  - Regressions: Continous labeling. This type of supervised learning is able to understand the correlation between data variables. For example, this is used for weather forecasting, where historical data is taken into account.
  - Classifications: Finite labeling. The training data is labeled with finite values, it is the simplest form of supervised learning. It classifies input data into a predefined classification.

  There are plenty of applications of supervised learning algorithms, such as fraud detection, recommendation solutions, risk analysis, etc.

- Unsupervised learning: Unsupervised learning processes unlabeled data. In these algorithms there are no training data nor supervisors, no human intervention at all. This can be due to the nature of the data itself, or to the lack of funds to pay for manual labeling. It is similar to supervised learning; it recognizes patterns, commonalities, and groups the data. They work iteratively and react on new pieces of

data depending on the presence or absence of these patterns found on previous data.

Therefore, unsupervised learning is still a challenge today. It is commonly used in business to understand and classify large volumes of unlabeled data, in some fields of statistics, mail filtering (spam-detecting), and even in social media to organize large amounts of information.

- Semi-supervised learning: This type of learning falls between supervised learning and unsupervised learning. The data is presented within a mix of labeled and unlabeled data. It is very common to have only a little part of the data labeled, and most of the data unlabeled. The goal of this learning is to make the algorithm to predict classes of future test data better than that from the model generated by using the labeled data, reaching a considerable improvement in learning accuracy.

  The acquisition of unlabeled data is cheaper than having all data labeled, so this is also an advantage.

  The applications of semi-supervised learning are, for instance, speech analysis, internet content classification (labeling webpages), and protein sequence classification (identify DNA strands).

- Reinforcement learning: It uses observations gathered from interaction with the environment to take actions through intelligent programs, or also called *agents*. The program is intended to perform a certain goal, without any help or teaching that guides it (no data at all).

  It is used to make someone learn to play to a game only by playing against an opponent or in autonomous vehicles too.

In Figure 1.2, it can be seen the sum-up of the machine learning types.

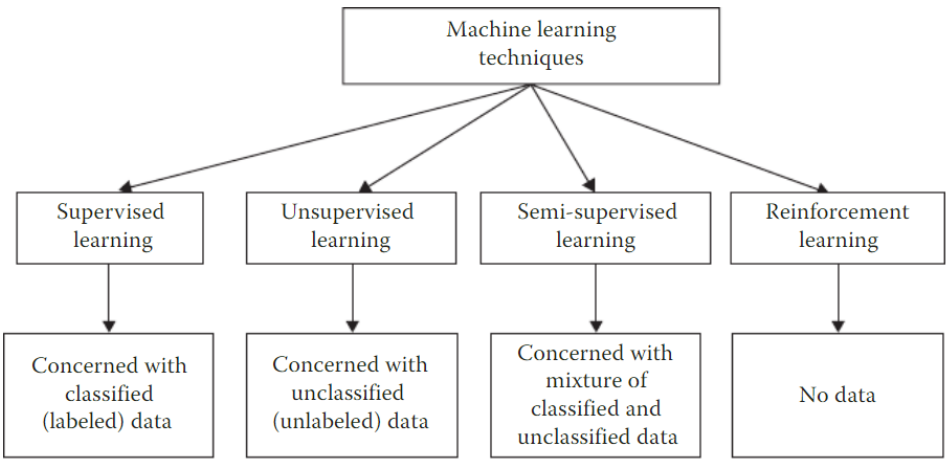


Figure 1.2: Machine learning types (Ref. [3])

There are plenty of important applications of these algorithms, and most of them used daily by people: recognition of handwritten digits, computer-aided diagnosis, computer vision, driverless cars, face recognition, speech recognition, text mining...

In this project we will focus on deep learning, a powerful branch of machine learning that is based on artificial neural networks. Deep learning is a sub-field of machine learning in Artificial Intelligence (AI) that deals with algorithms inspired from biological structure and fuctioning of a brain to aid machines with intelligence (Figure 1.3).

Deep learning was discovered around the 1980s, but it has not been until now that this field has gotten stronger. This is due to the large amount of data that nowadays companies handles, and also the powerful computers that we have. It has many applications, such as speed recognition, driverless cars, voice control, social network filtering, translation, medical image analysis, etc. Deep learning is a powerful set of techniques for learning in artificial neural networks, and there are different types of them. The most used artificial neural networks are the deep neural networks (DNN), convolutional neural network (CNN) and the regular neural network (RNN), put especially the first two mentioned. The following chapter details the first and simplest artificial neural network, the deep neural network (DNN).

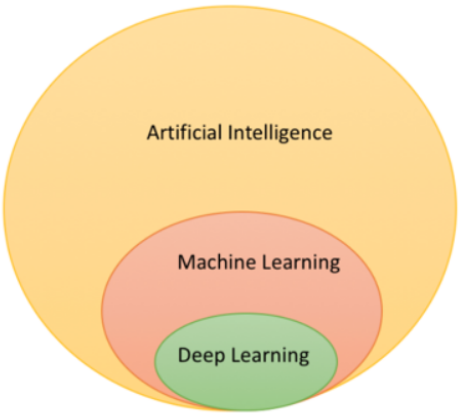


Figure 1.3: AI, machine learning and deep learning relation (Ref. [4])

CHAPTER 2. DEEP NEURAL NETWORKS

The second chapter is based on Deep Neural Networks, the first neural networks ever created. We are going to understand the concepts of a DNN, and focus on how to make a network learn. For that purpose, gradient descent and stochastic gradient descent are going to be explained in detail.

Furthermore, it is going to be understood how to interpret curves of learning, considering the worst cases that can occur: underfitting and overfitting. Finally, the statement of the practical part where a simple DNN algorithm is implemented.

2.1. Deep Neural Networks

Deep Neural Networks are the first and simplest type of artificial neural networks. They are also called Feedforward Networks, Fully Connected Networks, or Multi-Layer Perceptrons Networks (MLPs).

As said before, deep learning pretends to simulate a human brain through artificial neural networks. These are made of *perceptrons* (just as human brains are made of neurons), that are connected to other perceptrons.

Nowadays, perceptrons are not common to use in neural networks because there are newest artificial neurons that work more efficiently, but it is important to know what perceptrons are because they were the first artificial neurons created.

A perceptron takes several binary inputs, and produces a single binary output. In Figure 2.1, it can also be observed the similarity of the perceptron to the human brain neuron:

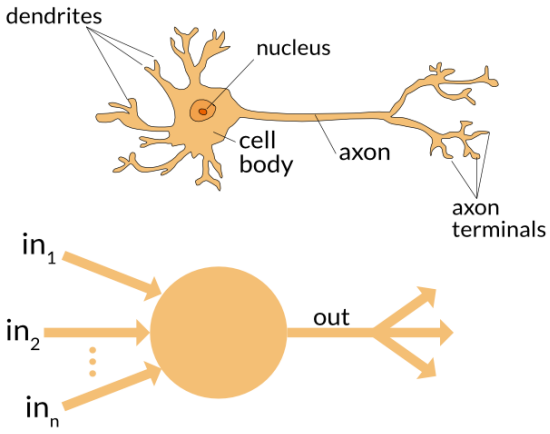


Figure 2.1: Perceptron vs. human brain neuron (Ref. [5])

Perceptrons join each other simulating human brain connections, and they are commonly represented as a hierarchical (layered) organization of perceptrons, named network (Figure 2.2). A network can be divided into three layers:

- Input layer: First layer of neurons. It is the leftmost layer of the network, and their neurons are called input neurons. They receive the inputs of the network.

- Hidden layers: Middle layer of neurons. The neurons are neither inputs nor outputs. Neural networks can have multiple hidden layers.
- Output layer: Last layer of neurons. It is the rightmost layer of the network, and their neurons are called output neurons. They produce the output or outputs of the network.

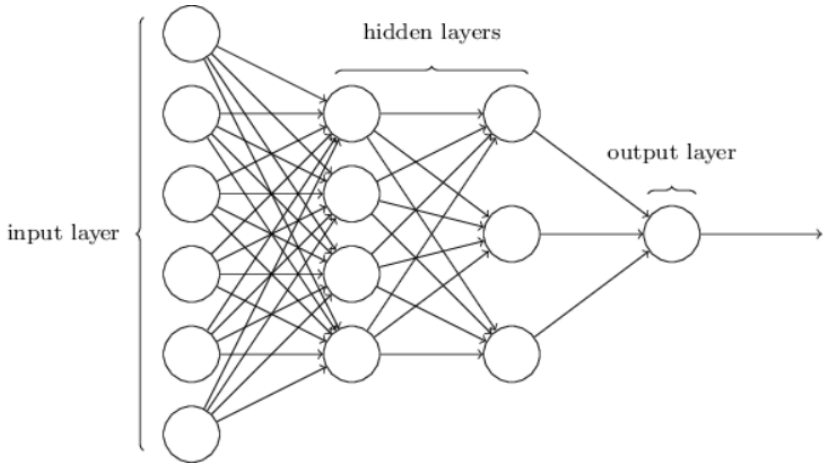


Figure 2.2: Artificial neural network layers (Ref. [6])

Therefore, the connections between neurons of successive layers have an associated *weight*, real numbers indicating the influence of the respective inputs to the output of the neuron, and helping the neuron on decision-making.

We can also define the *activation function*, a function that is computed for each neuron that defines the signal to pass to the next connected neurons. If the output of a neuron results in a value greater than a threshold, the output is passed, and if not it is not passed. If we define the inputs of the perceptron as x_1, x_2, x_3, being x_j the set of inputs with length j; and similarly with weights, being w_j the set of weights, we can define mathematically how a perceptron works as following:

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{thresold}, \\ 1 & \text{if } \sum_j w_j x_j > \text{thresold} \end{cases} \tag{2.1}$$

And, moving the threshold to the other side of the inequality, we replace it by the *bias* of the perceptron b = -threshold.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0, \\ 1 & \text{if } \sum_j w_j x_j + b > 0 \end{cases} \tag{2.2}$$

Being 0 or 1, if the perceptron's output excites or inhibits.

Then, we want the network to learn to solve a problem. This can be made by forcing our network to learn weights and biases so that the output is the desired one. This is only possible if we define a neuron that not only has 0 and 1 outputs, but any real value between them: the *sigmoid neuron*.

Sigmoid neurons are neurons similar to perceptrons, but sensitive to small changes in their weights and biases, that also produces small changes on the output. A sigmoid neuron is defined mathematically by the sigmoid function σ:

$$\sigma = \frac{1}{1+e^{-z}}. \tag{2.3}$$

Being $z = w \cdot x + b$. When z is very large and positive, the output of the neuron is approximately 1; and otherwise, the output is approximately 0. This turns the sigmoid neuron to be very similar to the perceptron.

σ is also a smooth version of a step function, which is the activation function of the perceptron (Figure 2.3). The smoothness shape of the sigmoid function is determinant to accomplish values from 0 to 1, and that makes the function to be commonly used in neural networks.

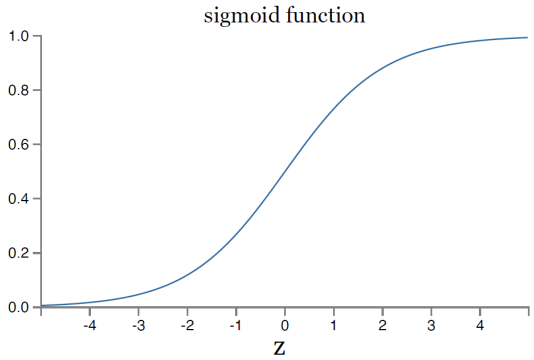


Figure 2.3: Sigmoid function (Ref. [6])

Something else is needed to make a DNN learn, a method that optimizes our algorithm and makes an every iteration update of the parameters of our model, like the gradient descent method.

2.2. Learning

In this section, learning methods in neural networks are going to be detailed. First learning with gradient descent, and then stochastic gradient descent learning technique. Finally, underfitting and overfitting cases are going to be determined as bad cases for the result on the learning process.

2.2.1. Learning with gradient descent

Gradient descent algorithm is an iterative optimizer method of first order that is used to minimize some function. In machine learning, is used to find the parameters of a model, the weights and biases that will make our network find the proper output for a given input.

In deep learning, that function is called cost function, quadratic cost function, or mean squared error (MSE). The point is to minimize it, as the function denotes how good our

model is; being 0 when the output is approximately equal to the input, and larger when it is not. Considering w and b as the weights and biases of the model, n as the total number of training inputs, x as the overall training inputs and a as the vector of outputs when the network has an input x, the *cost function* can be defined as:

$$C(w,b) \equiv \frac{1}{2n}\sum_{x}\|y(x)-a\|^2. \tag{2.4}$$

To explain it in a simpler way, we are going to imagine a function $C(v)$ of just two variables (it could be more) v_1 and v_2, like in Figure 2.4.

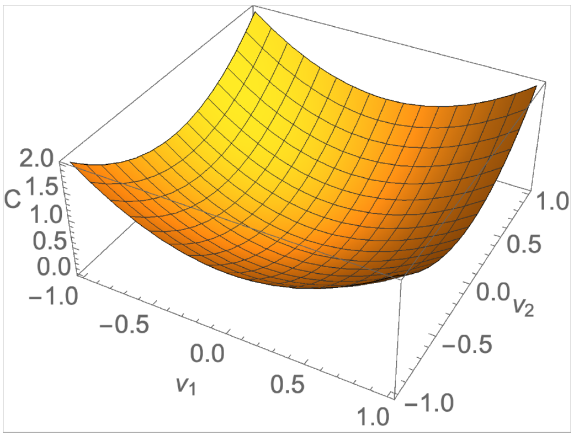


Figure 2.4: Function $C(v_1,v_2)$ (Ref. [6])

The point is to minimize the function, so we want C to achieve its global minimum. It is important to remark that this method is used only when an analytical manner is not possible, like this case. If you imagine a ball rolling down the function, the motion of the ball can be described as following:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{2.5}$$

Being Δv the amount of movement of the ball, and v the direction of the ball for variables 1 and 2.

Rewriting the equation considering that $\Delta v_1, \Delta v_2$ can be defined as Δv vector of changes in v $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, and that is also possible to define the gradient of C as $(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$, the equation can be rewritten as:

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{2.6}$$

It can be observed that gradient vector ∇C relates changes in v to changes in C, just as expected in a gradient. What is particular of this equation is that Δv is defined as:

$$\Delta v = -\eta \nabla C, \tag{2.7}$$

To make ∇C negative and succeed on our purpose. η is known as the *learning rate*, a small and positive parameter of the network. It can be defined as the following update rule as the gradient descent algorithm:

$$v \rightarrow v' = v - \eta \nabla C. \tag{2.8}$$

Gradient descent algorithm is a powerful way to minimize the cost function and also helping the network learn.

If this is applied to the particular case of a neural network, with weights w_k and biases b_l and the corresponding components of vector ∇C on these weights and biases, we have:

$$w_k \rightarrow w'_k = w_k - \eta \nabla \frac{\partial C}{\partial w_k} \tag{2.9}$$

$$b_l \rightarrow b'_l = b_l - \eta \nabla \frac{\partial C}{\partial b_l}. \tag{2.10}$$

With these iterations, we finally arrive at the minimum of the cost function C. However, it is important to remark that Equation 2.4 referring to the cost function iterations can last many time if the number of training inputs x is large, and then the learning is going to be slow too.

2.2.2. Learning with stochastic gradient descent

Stochastic gradient descent is a variation of the original gradient descent algorithm used to increase learning speed. Instead of making all the iterations to get ∇C gradient, the method estimates it by computing the gradient for a small sample of randomly chosen training inputs, named as *mini-batch*.

Therefore, the update rule is modified in the next way:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum \frac{\partial C_{X_j}}{\partial w_k} \tag{2.11}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum \frac{\partial C_{X_j}}{\partial b_l}. \tag{2.12}$$

Being m the number of mini-batch where the sums are overall the training samples X_j in that mini-batch. Furthermore, an *epoch* of training is completed when the training inputs are consumed. Then, a new training epoch initiates.

For instance, if we have a training set of size n=80,000, and a minibatch of size m=10 is chosen, this translates into a factor of 8,000 increase in speed estimating the gradient. We have to take into account that the estimation won't be perfect, there will be fluctuations, but there is no need for an accurate calculation of the ∇C gradient.

So stochastic gradient descent is a well-known method for learning in neural networks, that speeds up learning and produces nice results.

2.2.3. Results on learning process. Underfitting and Overfitting

A good machine learning model makes predictions with no error, it approximates/fits the data after the learning process. It generalizes any new input data from the learned domain in a proper way.

Achieving a good fit is not always easy, sometimes the algorithm is not well propounded and it is necessary to modify it. In this context, the terms overfitting and underfitting are introduced. The perfect fit is an intermediate between these two (Figure 2.5).

- Underfitting: A model that is not capable to find a pattern on data, it does not fit the data well enough. Usually it happens because there is not enough training data, it needs more training, or we are trying to construct a linear model from non-linear data. Due to his low accuracy, it makes errors in predictions.

  Methods to reduce underfitting:

  - Increase the time of training or the number of epochs of the model.
  - Increase training data.
  - Alternate types of machine learning algorithms or model complexity.
  - Remove noise from data.

- Overfitting: A model that learns too much about the training data set. Overfitting occurs when there is a lot of training data, or we spent many epochs/time learning. At this point, the model starts to learn of the noise and the details of training data, meaning that the model will not be capable to assume and make predictions on new data, because it is only capable to do it on the given training set. It has high accuracy but it makes errors on predictions.

  Methods to reduce overfitting:

  - Decrease time of training, or the number of epochs of the model.
  - Remove noise from data, decreasing the training data that is confusing.
  - Reduce the model complexity.
  - Use some type of regularization.

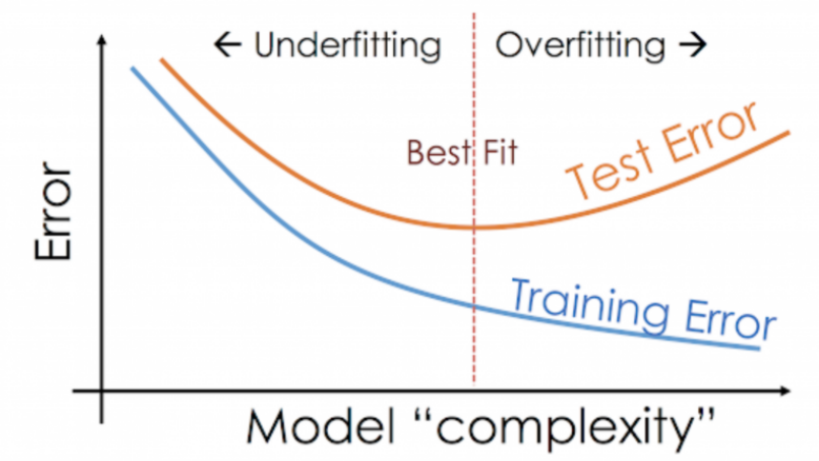


Figure 2.5: Curves of learning showing underfitting and overfitting (Ref. [7])

Both underfitting and overfitting show poor generalization to new data. The difference between them is that overfitting shows good performance on training data, and underfitting not even that. In conclusion, the goal is to achieve a trade-off between underfitting and overfitting, but it can be difficult in practice.

2.3. Example of a simple DNN

In this section, it is going to be explained the implementation of a simple Deep Neural Network (DNN) applied to the MNIST database.

2.3.1. MNIST database

The MNIST (Modified National Institute of Standards and Technology database) database is composed of a large collection of handwritten digits, that it is common to use for training images in processing systems, and on machine learning discipline. The objective is to classify a given handwritten digit image into 10 classes that represent the integer values from 0 to 9.

The database is a modified subset of two data set made by NIST (United States National Institue of Standards and Technology), specifically the Special Database 1 and Special Database 3. These databases were created scanning handwriting samples from 250 people, half of the US Census Bureau employees, and a half from high school students.

The MNIST dataset is divided into two parts. The first part corresponds to training data, which is composed of 60,000 training images; and then, the second part corresponding test data contains 10,000 testing images, that are used to verify how good the model is at recognizing new data. Both test and training images were collected by half employees, and half high school students, as mentioned above. These images were normalized in 28x28 greyscale images (Figure 2.6).

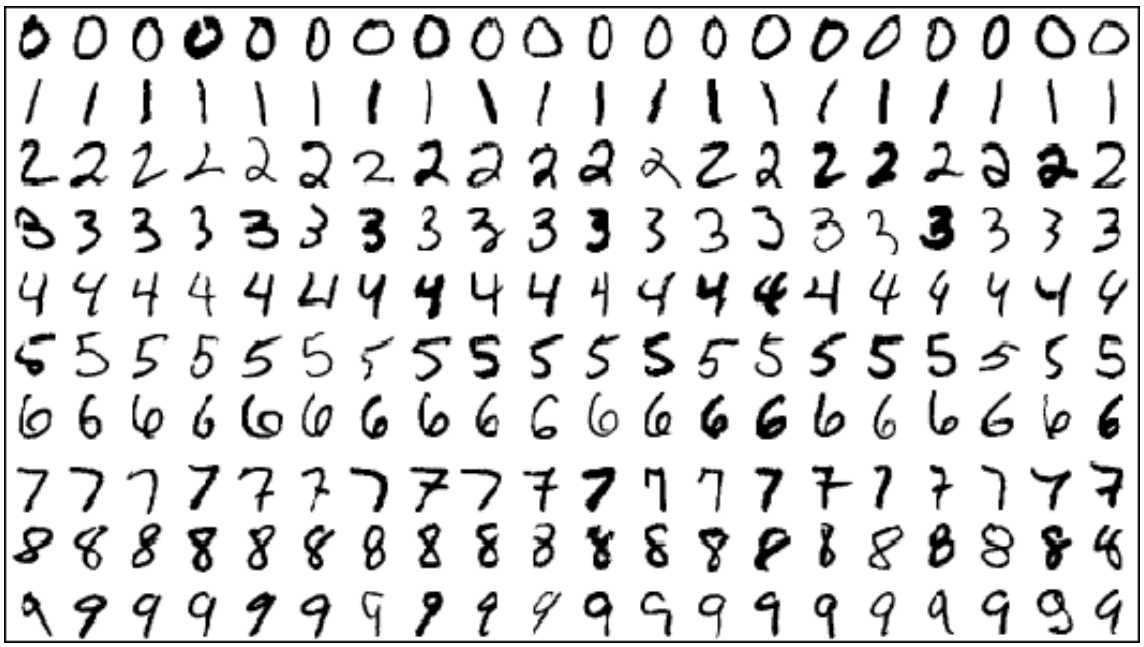


Figure 2.6: Images subset of the MNIST database (Ref. [6])

It is a nice database to initiate in the field of deep learning because it is not necessary to spend efforts on preprocessing and formating. It is a free-source easy to find on the Internet.

It is important to remark that some researches have achieved “near-human performance” in this database. In fact, in 2018 researchers from the Department of System and Information Engineering from the University of Virginia, succeed with a 0.18 error only, using simultaneous three kinds of neural networks.

2.3.2. Analyzing the MNIST database with a simple DNN

The practical task done in this chapter is the construction of a simple Deep Neural Network (DNN) on the MNIST database.

A machine learning algorithm has been implemented to make a neural network learn to identify digits from 0 to 9 using the MNIST database explained in the previous subsection 2.3.1. of the present chapter.

The algorithm is adapted from the book Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015. It is an online book that explains clearly the concepts of Neural Networks and Deep Learning from the very beginning of it. Moreover, the algorithm is constructed in Python 3.8 language and implemented in a web-based interactive development environment named Jupyter notebook. Full code is in Appendix A.

The code is composed of the implementation of different classes. The core class is the *Network* class, the one that has been modified and adapted for making this task. The Network class represents the neural network itself, made of functions related to the concepts explained during this chapter. The most important methods and parts of the full code are explained in the next lines.

To initialize the DNN, the code used is in Figure 2.7:

```
class Network(object):

    def __init__(self, sizes):

        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

Figure 2.7: Intializing the network (Own elaboration)

The list *sizes* contains the number of neurons in the respective layers of the network. From that list is that the network can be initialized, with biases and weights randomly initiated with a Gaussian distribution with 0 mean, and variance 1.

Furthermore, the implementation of the *feedforward* method is also important, because, given an input “a” for the network, it returns the corresponding output (Figure 2.8).

The most important thing for our network is learning. Thus, this is executed by implementing the *SGD* (stochastic gradient descent) function, which trains the network using mini-batch stochastic gradient descent. Training data is collected in a list of tuples (x, y) with the training inputs and the desired outputs, respectively. Then, test data will evaluate the network after each epoch, with partial progress printed out. This can be observed in Figure 2.9.

```
    def feedforward(self, a):

        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

Figure 2.8: Feedforward method (Own elaboration)

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):

        training_data = list(training_data)
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
```

```
                print("Epoch {} : {} / {}".format(j,self.
↪evaluate(test_data),n_test));
            else:
                print("Epoch {} complete".format(j))
```

Figure 2.9: Stochastic gradient descent learning (Own elaboration)

Finally, it is important to emphasize the *update mini-batch* method, that updates the neural network weights and biases according to a single iteration of gradient descent, using the training data in that mini-batch (Figure 2.10).

```
    def update_mini_batch(self, mini_batch, eta):

        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

Figure 2.10: Update mini batch method (Own elaboration)

How well does the algorithm recognize handwritten digits? This can be known by executing a set of commands, where we are going to load the MNIST database, and also defining the parameters of the network we want to implement (Figure 2.11).

```
import mnist_loader
```

```
training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

```
import network
```

```
net = network.Network([784, 30, 10]) # A network with first layer (784␣
 ↪neurons),30 hidden neurons and 10 neurons in the third layer.
```

```
network=net.SGD(training_data, 30, 10, 3.0, test_data=test_data) # Learn with␣
 ↪30 epochs, with a mini batch size of 10, and a learning rate 3.0.
```

Figure 2.11: Set of commands to make our network learn (Own elaboration)

In the first two lines of code, the MNIST database is loaded, and then, class Network is imported to define the neural network wanted to learn. In this case, the network has 784 neurons in the first layer, 30 neurons in the hidden layer, and 10 neurons in the output layer. To conclude, the network is trained by the stochastic gradient descent (SGD) method in 30 epochs, a 10 mini-batch size, and a learning rate η of 3.0.

The output of the algorithm can be observed in Figure 2.12:

```
Epoch 0 : 8284 / 10000
Epoch 1 : 8353 / 10000
Epoch 2 : 8452 / 10000
Epoch 3 : 8460 / 10000
Epoch 4 : 9269 / 10000
Epoch 5 : 9376 / 10000
Epoch 6 : 9433 / 10000
Epoch 7 : 9406 / 10000
Epoch 8 : 9413 / 10000
Epoch 9 : 9407 / 10000
Epoch 10 : 9444 / 10000
Epoch 11 : 9426 / 10000
Epoch 12 : 9451 / 10000
Epoch 13 : 9454 / 10000
Epoch 14 : 9429 / 10000
Epoch 15 : 9459 / 10000
Epoch 16 : 9468 / 10000
Epoch 17 : 9458 / 10000
Epoch 18 : 9470 / 10000
Epoch 19 : 9496 / 10000
Epoch 20 : 9472 / 10000
Epoch 21 : 9436 / 10000
Epoch 22 : 9490 / 10000
Epoch 23 : 9477 / 10000
Epoch 24 : 9483 / 10000
Epoch 25 : 9483 / 10000
Epoch 26 : 9478 / 10000
Epoch 27 : 9502 / 10000
Epoch 28 : 9507 / 10000
Epoch 29 : 9498 / 10000
```

Figure 2.12: Output of the algorithm (Own elaboration)

Where it can be observed that after 30 epochs, an accuracy of 94.98% is achieved. It is

important to consider that this is a nice result considering the simplicity of the algorithm. Thus, we consider a failure of 5.02%.

CHAPTER 3. CONVOLUTIONAL NEURAL NETWORKS

In this chapter, Convolutional Neural Networks are presented, explaining in detail the most important layers that exist in this type of network. Next, dropout and batch normalization regularization techniques are going to be described too. Finally, the practical part attaches an implementation of a simple CNN based on the MNIST database and a more complex CNN based on the CIFAR-10 dataset.

3.1. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are one of the most used neural networks nowadays. They are widely used for image classification, image recognition, face identifying, object detection, etc. The particularity of CNN's, is that they assume the inputs as images, which modifies in a certain way the architecture of the network, and hence it simplifies the parameters of our algorithm.

Why don't we use Deep Neural Networks? Imagine a normal size image, bigger than MNIST images, for instance, 300x300x3 (200 wide, 200 high, 3 color channels). DNN, or Fully Connected Networks, would lead the neurons to have 300*300*3=270,000 weights. That would be a time-waste of iterations and iterations, and a huge number of parameters that would lead the network to overfit. So, it is inconceivable that a Fully Connected Network can manage normal size images tasks. This requires a different network, CNN.

Unlike DNN, Convolutional Neural Networks layers are adjusted in 3 dimensions: width, height, depth. The concept is that the network is capable to transform a 3D input volume to a 3D output volume through his layers specifically constructed for that purpose (Figure 3.1).

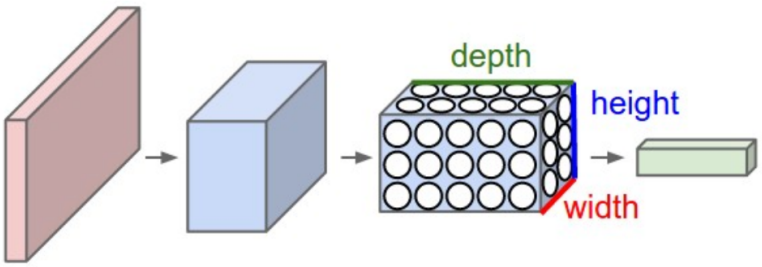


Figure 3.1: CNN structure (Ref. [8])

A CNN is made of a sequence of layers, being the most popular layers the Convolutional Layer, Pooling Layer, and Fully-Connected Layer, that will compound the architecture of the network. It is also important the dimensions of each layer, representing their function on the network, and that some layers own parameters and others don't. Simple architecture for a convolutional network can be described as the following, assuming, for instance, an input image 64x64x3:

- Input *(INPUT)*: First layer composed of the pixel matrix values of the corresponding image. [64x64x3]

- Convolutional Layer *(CONV)*: Layer that makes the convolution between the filters and local regions of the input image. [64x64x10] (10 filters used)

  This is the layer that makes most of the calculation of the network. As mentioned before, full connectivity implies time and it is wasteful for this type of network. Convolutional layer introduces the connection to a local region of the input volume, which produces an activation map of every connection made. These small regions are called *filters* that perform dot products between the entries of the filter and the input of the local regions of the image when moving across their dimensions (width and height). Eventually, filters will learn when they observe the same pattern on activation maps of different images.

  The output of this layer is the set of 2D activation maps. We can observe how a convolutional layer makes its calculations in Figures 3.2 and 3.3:

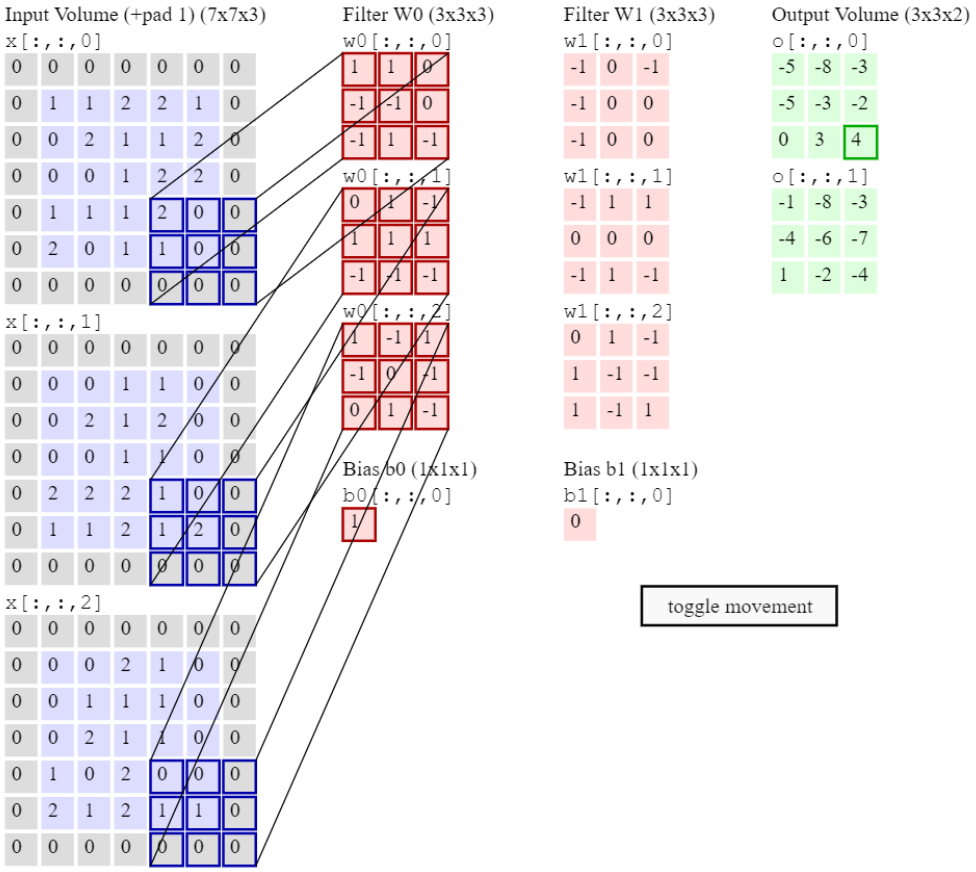


Figure 3.2: Convolutional layer (Ref. [8])

- Rectified Linear Unit *(RELU)*. Relu stands for Rectified Linear Unit, that implements the activation function $max(0,x)$. [64x64x10]

  RELU layer activation function behaves linearly given values greater than zero, but it is a nonlinear function if the values are negative (Figure 3.4). In that case, the values are converted into a zero.

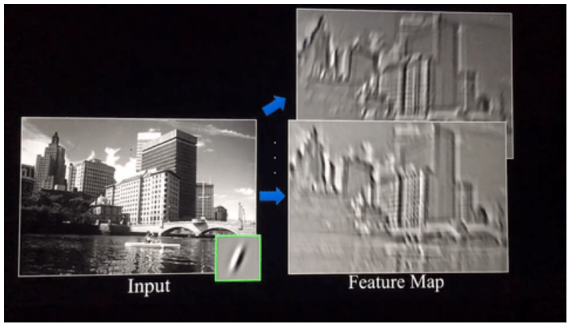


Figure 3.3: Visual convolutional layer (Ref. [9])

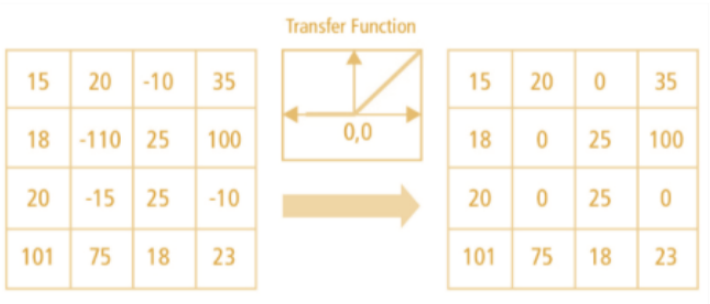


Figure 3.4: RELU layer (Ref. [10])

- Pooling Layer *(POOL)*. Layer that executes a downsampling action along the three dimensions. [32x32x10]

  The pooling layer reduces the number of parameters in images by making downsampling. That results in a progressive reduction of the spatial size of the images. There are different functions to implement pooling, for instance:

  - Max Pooling
  - Average Pooling
  - Sum Pooling

  The most used method is the max pooling, which takes the largest element from the rectified feature map applied by a filter. This layer is very important because it helps to reduce overfitting. The effect produced by the layer can be observed in Figure 3.5.

- Fully Connected Layer *(FC)*. Layer that calculates the class scores. [1x1x10]

  This layer is fully-connected, as a DNN. The concept is to flatten the matrix from the previous layer and put it into a fully connected layer process. Finally, in this layer, we have an activation function called *softmax*, which classifies the outputs into the corresponding classes based on the scores acquired.

An example of an implemented CNN can be seen in Figure 3.6.

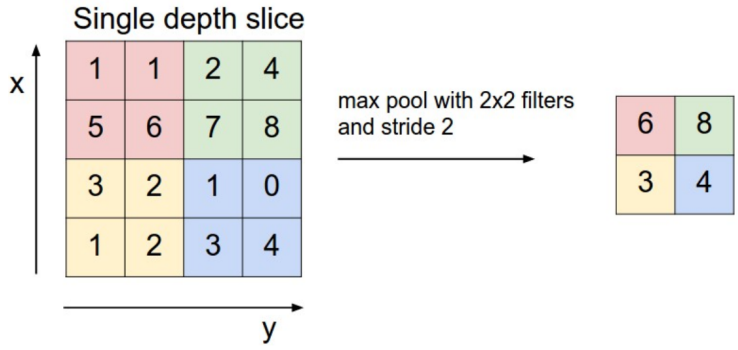


Figure 3.5: Max pooling (Ref. [8])

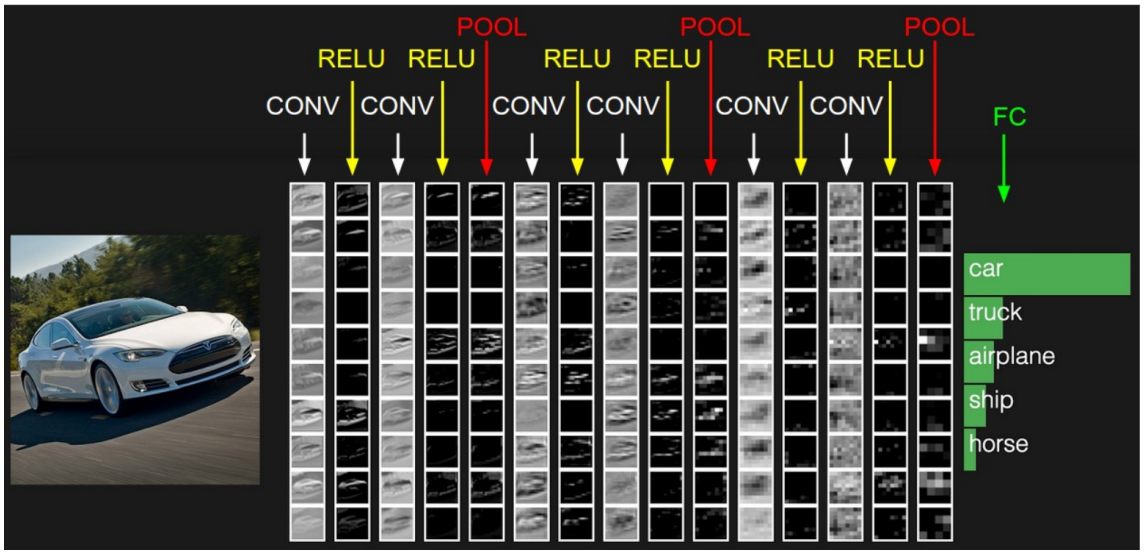


Figure 3.6: Example of CNN (Ref. [8])

3.1.1. Regularization techniques

In machine learning, regularization is a way to prevent overfitting and co-dependency amongst neurons during training. Regularization reduces it by making slight modifications to the algorithm, for instance removing some neurons on the layers for better performance. The model is trained such that it does not learn an interdependent set of features weights, and it is capable to generalize data better.

There are many types of regularization techniques, but the most popular ones are the following:

- *Dropout*: Dropout is a regularization technique that prevents complex co-adaptations on training data, it "ignores" neurons randomly during the training phase, as observed in Figure 3.7. It is a very efficient way of performing model averaging with neural networks.
- *Batch Normalization*: Batch normalization is a regularization technique that standardizes the inputs to a layer. It normalizes the output of the previous layer by

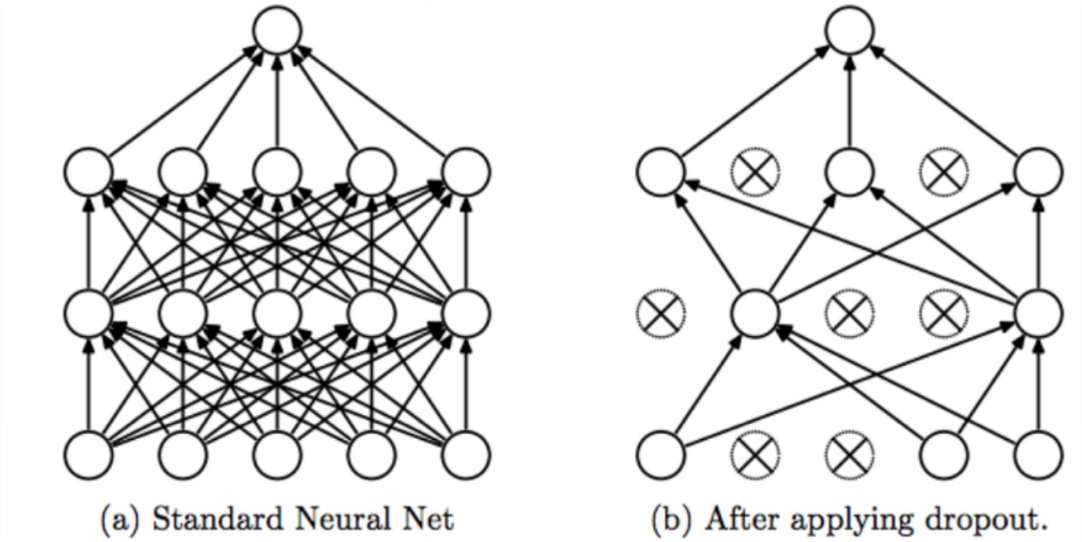


Figure 3.7: Dropout technique (Ref. [11])

subtracting the batch mean, and then dividing by the batch variance. It produces a notorious acceleration on the training process of the network, and eventually, it improves the performance of the model.

In Figure 3.8, it can be seen the mathematical process of batch normalization on a mini-batch:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 3.8: Batch Normalization process (Ref. [12])

3.1.2. Keras: the Python deep learning API

Keras is a free application programming interface (API) that supports artificial neural networks (Figure 3.9). It works in Python language, and it runs on top of the machine learning platform Tensorflow 2.0, which is an open-source machine learning platform.

Keras is a relatively new tool that was created in 2015 by a Google engineer named François Chollet. It is a platform that executes low-level tensor operations on CPU, GPU, or TPU. Keras allows us to design executable models on smartphones (iOS and Android), and also on the web.

Keras provides an efficient, highly-productive interface for solving deep learning problems, supporting different types of neural networks.

Figure 3.9: Keras platform (Ref. [13])

3.2. Analyzing the MNIST database with a simple CNN

One of the two practical tasks made in this chapter is the construction of a simple Convolutional Neural Network (CNN) on the MNIST database (2.3.).

Unlike the previous implementation of a DNN, the CNN is adapted from `https://keras.io/examples/mnist_cnn/` Keras webpage. As mentioned in the previous subsection, Keras uses Python 3.8 language. The code is implemented on a Jupyter Notebook environment, and full code can be found in Appendix B.

The code is written in a single *.ipynb* file, where all the structure and execution of the neural network algorithm is implemented. The most important parts of the code are explained in the next lines:

To use Keras environment first is needed to import dependencies of different features, including the network layers that are going to be used (Figure 3.10).

Then, we define the Convolutional Neural Network parameters that are going to be used at the algorithm, as seen in Figure 3.11. As it is expected, if the number of epochs is bigger, the learning time will be too; and the same will happen with the size of the mini-batch. Keras default parameters have been set in this step; being a mini-batch size of 128 and 12 epochs.

Unlike the practical DNN realized in section 2.3.2., the learning model is saved due to the *os* dependency, which allows the user to save it in a chosen directory with a given input name (Figure 3.12).

```
from __future__ import print_function
import keras # Import keras deep learning library.
from keras.datasets import mnist # Import the MNIST dataset from keras.
from keras.models import Sequential # The simplest and more friendly model to␣
 ↪program keras and define the layer's structure.
from keras.layers import Dense, Dropout, Flatten # Import the type of layers␣
 ↪that will be used from keras.
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.preprocessing.image import load_img, img_to_array # To test the CNN␣
 ↪and rescale an entering image.
import os # For looking and directories and import settings.
import matplotlib.pyplot as plt
import timeit # We import it to calculate how many time lasts our models to be␣
 ↪trained.
```

Figure 3.10: Import dependencies (Own elaboration)

```
batch_size = 128
num_classes = 10 # The 10 digits 1,2,3,4,5,6... collected in 10 clases.
epochs = 12
```

Figure 3.11: Define network parameters (Own elaboration)

```
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'mnist_cnn_trained.h5'
```

Figure 3.12: Save the model (Own elaboration)

Likewise, the MNIST dataset is loaded providing the training data and test data vectors. Furthermore, the architecture of the network is defined using Keras layers (Figure 3.13). In this example, we use two convolutional layers, a max-pooling layer, and we use the dropout regularization technique by default on Keras. Finally, the fully-connected layer is implemented with the softmax function.

```
model = Sequential() #We import the sequential model of keras.

#Next we describe the arquitechture of the CNN (layers).
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Figure 3.13: Arquitecture of the CNN (Own elaboration)

Finally, the Convolutional Network is trained and the final accuracy achieved is 99.15 %, and a loss of only 0.0281. The algorithm saves the trained model and the model accuracy and loss are plotted, respectively (Figure 3.14 and Figure 3.15).

As can be seen, the accuracy of 99.15% is a very nice result. Learning curves show a good fit for the data.

```
Epoch 10/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0315 -
accuracy: 0.9908 - val_loss: 0.0294 - val_accuracy: 0.9912
Epoch 11/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0287 -
accuracy: 0.9913 - val_loss: 0.0269 - val_accuracy: 0.9917
Epoch 12/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0272 -
accuracy: 0.9915 - val_loss: 0.0281 - val_accuracy: 0.9915
Time:  1316.3387901000096
Saved trained model at C:\Users\claud\OneDrive\Escritorio\TFG\TFG\JupyterNoteboo
k\saved_models\mnist_cnn_trained.h5
```

Figure 3.14: Training of the CNN (Own elaboration)

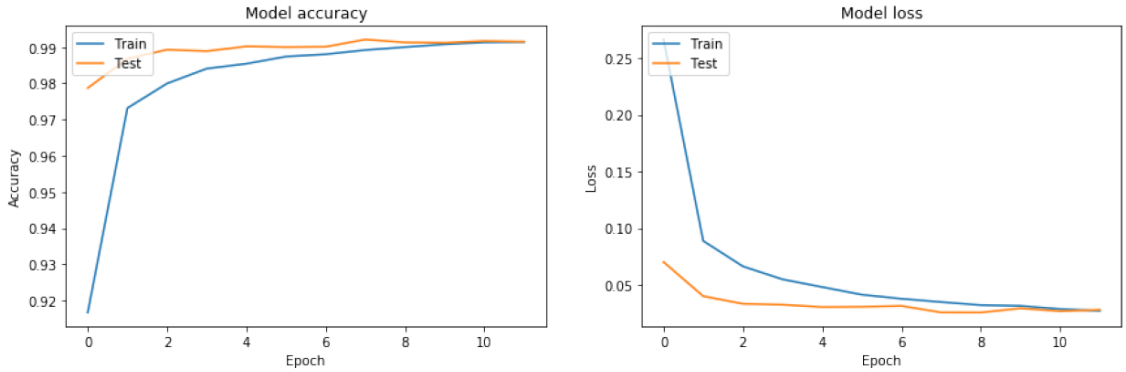


Figure 3.15: Learning curves of the CNN model (Own elaboration)

3.3. Example of a CNN

In this section, it is going to be presented the implementation of a Convolutional Neural Network (CNN) applied to the CIFAR-10 dataset.

3.3.1. CIFAR-10 dataset

The CIFAR-10 (Canadian Institute For Advanced Research) dataset is a collection of images commonly used in machine learning for computer vision and image processing training (Figure 3.16). Images were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It consists of 60,000 32x32x3 images, divided in 50,000 training images and 10,000 test images.

The images represent 10 different classes, where each of them is represented by 6,000 images. It is important to remark that the classes are mutually exclusive, the same photo can't be in more than one class. The 10 classes of CIFAR-10 are the following:

- Airplane
- Automobile
- Bird
- Cat

- Deer
- Dog
- Frog
- Horse
- Ship
- Truck

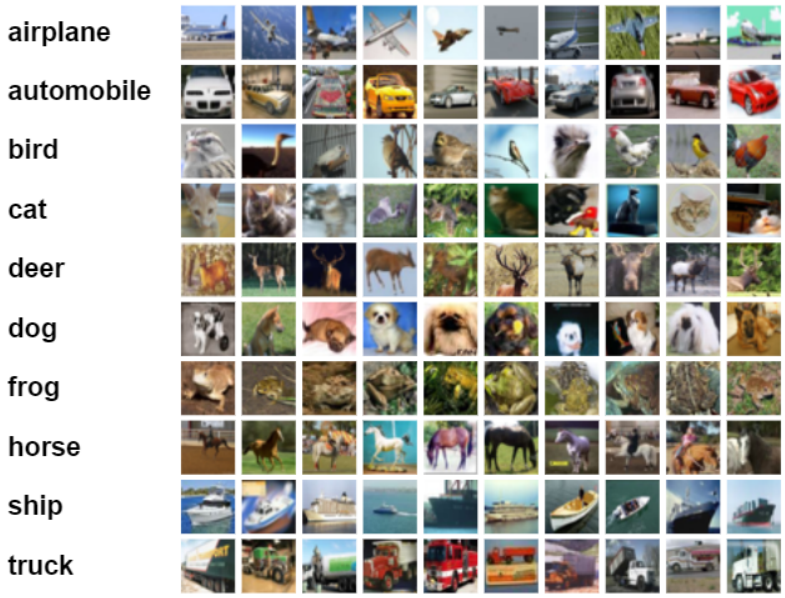


Figure 3.16: CIFAR-10 dataset (Ref. [14])

This dataset has historically achieved an error of only 3.47%.

Like the MNIST database, CIFAR-10 is also a free source database that can be downloaded from the Internet. There exists an improved variant of the CIFAR-10 dataset, the CIFAR-100 dataset where there are 100 classes with 600 images each. We have decided to implement the CIFAR-10 because it fits better for our academic implementation.

3.3.2. Analyzing the CIFAR-10 dataset with a CNN

The final task done on this set of implementations of neural networks is the a CNN algorithm on the CIFAR-10 dataset. In the same way as the previous implementation, the algorithm made in this part of the project is adapted from Keras webpage https://keras.io/examples/cifar10_cnn/, so the language is the same (Python 3.8), and it is also implemented into a Jupyter Notebook environment. Full code can be found in Appendix C.

The notebook considers five cases of implementation of the CNN, that will be compared in order to obtain the better performance of the neural network. The five trained models are characterized by the following features:

1. Simple model using stochastic gradient descent optimizer (SGD)
2. Simple model using rmsprop optimizer
3. Model using rmsprop optimizer and dropout regularization technique
4. Model using rmsprop optimizer and batch normalization technique
5. Model using rmsprop optimizer and both regularization techniques (dropout and batch normalization)

They will be presented from the simplest (model 1) to the most complex model (model 5). Finally, we will be able to find the best implementation for our needs.

The algorithm architecture is similar to the previous model of CNN on the MNIST database. The difference is the loading of the CIFAR-10 dataset, and also the implementation of different layers and techniques used along with the five models. Moreover, the mini-batch size has been 32 and 10 number of epochs (Figure 3.17).

```
model = Sequential() #We import the sequential model of keras.

#Next we describe the arquitechture of the CNN (layers).
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
if dropout:
    model.add(Dropout(0.25))


model.add(Conv2D(64, (3, 3), padding='same'))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
if dropout:
    model.add(Dropout(0.25))


model.add(Flatten())
model.add(Dense(512))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
if dropout:
    model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Figure 3.17: Arquitecture of CIFAR-10 CNN (Own elaboration)

This architecture includes more layers than the previous one, for instance, Convolutional Layers and RELU layers that achieve better results on the network. Furthermore, different techniques are applied based on the model chosen by the user at the beginning of the notebook. It is important to remark that rmsprop optimizer is an improved version of SGD optimizer, using and adaptative learning in every step.

The importance of this study relies on the comparison of the results of the five trained models. Again, all code implemented can be found in Appendix D.

To determine if all models fits on data and also on new data, it is necessary to compare the learning curves (accuracy and loss curves) in plots, showed in Figures 3.18 and 3.19:

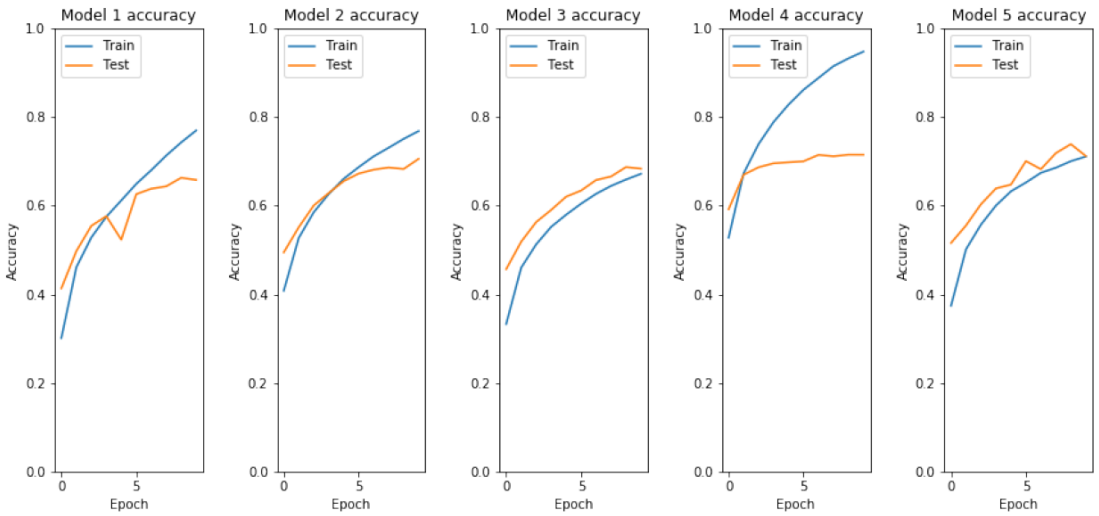


Figure 3.18: Accuracy of the 5 models implemented (Own elaboration)

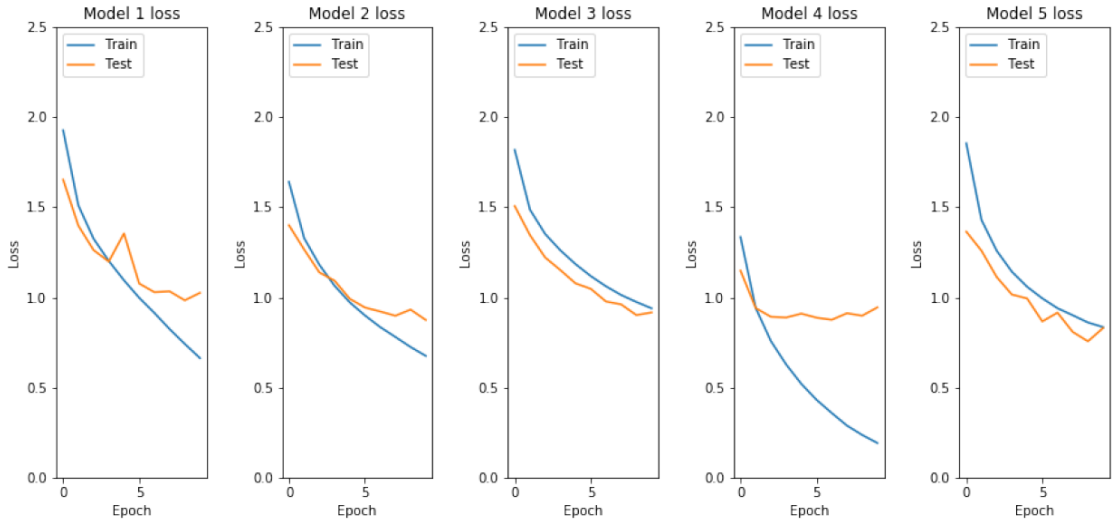


Figure 3.19: Loss of the 5 models implemented (Own elaboration)

Being the final accuracy and loss values of the models:

1. 65.84 % accuracy and 1.03 loss

2. 70.59 % accuracy and 0.87 loss
3. 68.37 % accuracy and 0.92 loss
4. 71.52 % accuracy and 0.94 loss
5. 71.23 % accuracy and 0.83 loss

The accuracies of the five models own high values, but that does not mean that models fit well the data. About the learning curves, it can be observed that we can discard, at first, three of the five models due to overfitting of the network. Models 1, 2, and 4 are overfitted because the loss curve show an inflection point where the validation curve overcomes the training curve. This could be modified by reducing the complexity of the algorithm or stopping the learning earlier (but it is important to remark that 10 epochs it is not a high number of epochs to learn).

Sometimes the best option is not to use all techniques at the same time, because this can produce overfitting by making the network learn the noise in training images.

So, in this situation, we have to compare the performance of model 3 and 5. Both accuracy values are great, being the highest the model 5 value. The same happens with loss values, the smallest one is also from model 5, but it is neither a great difference to appreciate. In addition, it can be taken into account the time spent on learning of the two models (Figure 3.20).

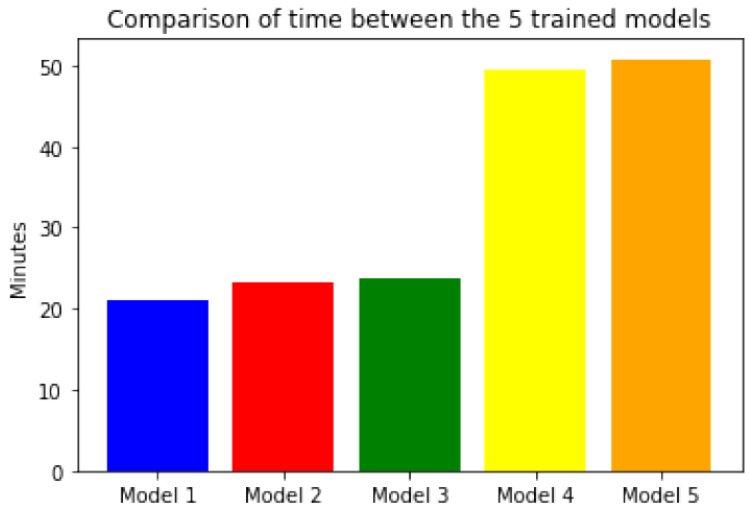


Figure 3.20: Time of learning of the 5 models implemented (Own elaboration)

Being the time of each model:

1. 21 minutes
2. 23 minutes
3. 24 minutes
4. 49 minutes

5. 50 minutes

Finally, for our purpose, we decide that the model best fit is model number 3. Number 3 model implements a CNN with rmsprop optimizer and dropout regularization technique. This model achieves a nice accuracy and loss values, it makes good predictions on new data, and it lasts an acceptable amount of time, 24 minutes, instead of the 50 minutes of model 5.

In Figure 3.21, we can observe a prediction of model 3 on a new data image from an airplane, imported from an specific directory.

Make a prediction:


```
|: # It can be any dimensions input image. The function "classify"
   #reshape it and make the prediction.
   classify('airplane1.jpg')

   airplane
```

Figure 3.21: Prediction of model 3 on an airplane image (Own elaboration)

It is remarkable that depending on the objective of the user, the chosen model can be different. Maybe time is very important for an specific labor, but maybe the most important thing is to achieve the largest accuracy value for the model. In this case, as it is an academic assignment, the model best fits the task is model number 3, as we look for an equilibrium of all features.

CHAPTER 4. OBJECT TRACKING

In this chapter I am going to expose my last practical task of the present project, an implementation of an object tracking algorithm using Detectron2. Firstly, the environment used is going to be detailed, and then the implementation and the results are going to be presented.

4.1. Object Tracking with Detectron2

Detectron2 is an open-source research platform from Facebook AI Research (FAIR) that implements object detection and segmentation (Figure 4.1). Is a modular object detection library that is based on PyTorch 1.6, a fast and effective tensor library used for deep learning and calculations in a GPU or CPU environment.

Detectron was born in 2018, and since that moment from now, it has become one of the most widely adopted open-source projects. Detectron has changed a lot during the last two years, implementing new tasks as semantic segmentation and panoptic segmentation.

It provides an intuitive, easy programming environment that allows researchers to design new models and make experiments from it. It is implemented in a modular design that allows users to modify each module independently, making a more efficient and flexible algorithm. It is scalable and it makes calculations very fast, and it also provides different recognized datasets.

Figure 4.1: Detectron2 logo (Ref. [15])

Detectron2 includes implementations for the following object detection algorithms: Mask R-CNN, RetinaNet, Faster R-CNN, RPN, Fast R-CNN, TensorMask, PointRend, DensePose, and more. It supports object detection with drawn boxes and instance segmentation masks, semantic segmentation, and panoptic segmentation. Some examples from Detectron2 can be observed in Figures 4.2 and 4.3.

The platform is distributed in GitHub, the world's leading software development platform. There, all the features like installation, start, licenses, and code of Detectron2 can be found. If you want to skip all the installation process, you can open a Detectron2 Tutorial file at Google Colab Notebooks, an online virtual code environment mostly used for IA algorithms, that allows free execution and programming in Python language, so as easy access to GPU.

Referring to object detection classes, Detectron2 implements the COCO dataset (Figure 4.4) by default. COCO dataset (Common Objects in Context) is a dataset that comprises 80 classes, with a total of 330,000 images, and it allows different segmentations. COCO dataset belongs to Microsoft, and it has collaborations of Facebook and companies that invest in AI.

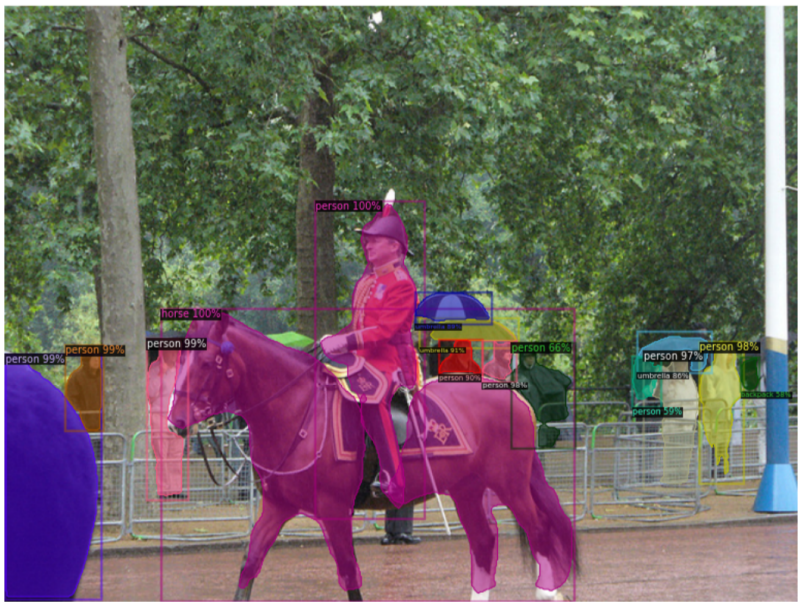


Figure 4.2: Object detection by Detectron2 (Ref. [16])

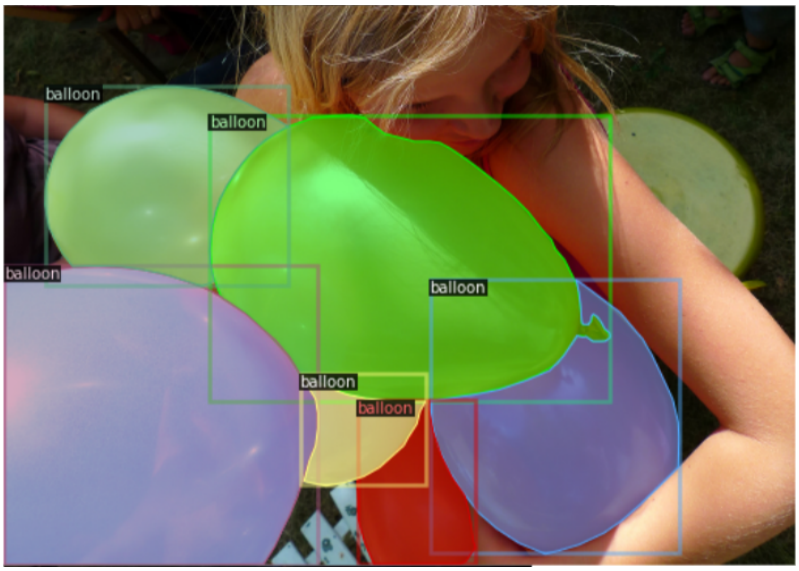


Figure 4.3: Object detection by Detectron2 (Ref. [16])

4.1.1. Implementing an Object Tracking method to Detectron2

As can be seen in the previous section, Detectron2 provides an environment with different object detection algorithms that can be implemented. Now, the objective is slightly different.

The purpose is to make an object tracking algorithm with the aid of Detectron2, an advanced machine learning software. The object tracking task consists of getting coordinates of particular objects detected by Detectron2 on given video frames, and study the motion of those objects. The outputs will be the total distance of the movement, the trajectory, velocity, and acceleration plots of the detected objects.

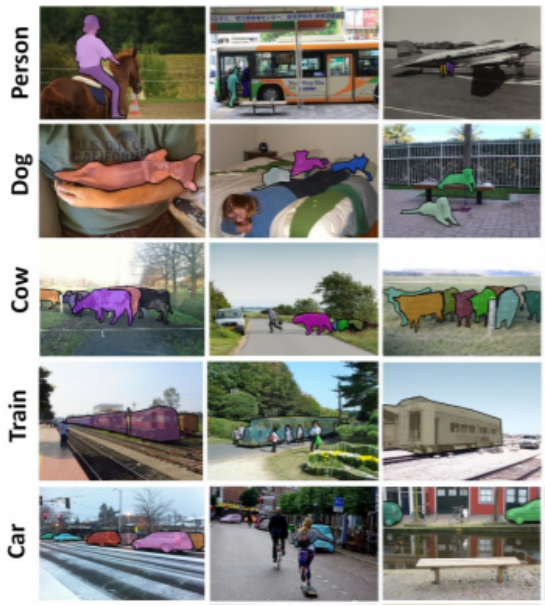


Figure 4.4: COCO dataset (Ref. [17])

To make it possible, the input of the program is a collection of frames from a video saved in our personal Google Drive environment. As Detectron2, the programming language has been Python 3.8 on a Google Collab environment.

The scenario of the video has been the following: I made an animation on PowerPoint that simulates two moving airplanes. The first airplane (airplane 0) is making a take-off maneuver, and the second airplane (airplane 1) is taxiing. In the first place, the purpose was to analyze a Youtube video of a static camera of a determined airport, but finally we chose to analyze a simpler 2D video, due to the difficulties that will be detailed later on section 4.1.2..

The animation has been implemented considering that an airplane taking-off is moving at a velocity between 70 m/s (252 km/h) and 80 m/s (288 km/h), and an airplane taxiing is moving around 15 m/s. Then, 50 frames of the video spaced 0.1 seconds were taken to be the input of the algorithm in Colab, being five seconds the total length of the video that has been analyzed. All code implemented can be found in Appendix E.

To justify the implementation, the most important parts of the algorithm are commented in this section.

First, we have to install Detectron2 on our Colab environment, and all the dependencies required. This implementation can be found in Figure 4.5.

Then, we import the images from our Google Drive and we save them into *images* vector (Figure 4.6).

When we have the vector with the corresponding images, we use the methods of Detectron2 to make the predictions on each image (Figure 4.7). On this step, we introduce the implementation of the *outputs[i]["instances"].pred_boxes.get_centers()* command to get the coordinates of the different objects detected on each image, making the *centers* tensor that contains all this information. We also do the same with the *outputs[i]["instances"].pred_boxes*, that provides the limits of the boxes drawn by the image. This will help us later to define

```
# install dependencies:
!pip install pyyaml==5.1 pycocotools>=2.0.1
import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version
# opencv is pre-installed on colab
```

```
# install detectron2: (Colab has CUDA 10.1 + torch 1.6)
# See https://detectron2.readthedocs.io/tutorials/install.html for instructions
assert torch.__version__.startswith("1.6")
!pip install detectron2 -f https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.6/index.html
```

```
# Some basic setup:
# Setup detectron2 logger
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import os, json, cv2, random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
```

Figure 4.5: Installation of Detectron2 (Own elaboration)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
# import the images from Google Drive
i=0
images=[]
for i in range(50):
    filename = str(i) + ".PNG"
    im=cv2.imread("/content/drive/My Drive/"+filename)
    images.append(im)
```

Figure 4.6: Import the 50 frames from Google Drive (Own elaboration)

the scale between the pixels of the images and the reality.

```
#outputs of learning
import tensorflow as tf
import math

#get instances and boxes predictions
centers=[]
heights=[]
for i in range(len(images)):
   outputs[i]["instances"].pred_classes
   outputs[i]["instances"].pred_boxes
   heights.append(outputs[i]["instances"].pred_boxes)
   heights[i]=np.array(heights[i])
   centers.append(outputs[i]["instances"].pred_boxes.get_centers())
   centers[i]=centers[i].cpu()
   centers[i]=centers[i].numpy()
   centers[i]=centers[i].astype(int)
   centers[i]=np.array(centers[i])

centers=np.array(centers) # contains the coordinates (x,y) of the centers of the boxes
heights=np.array(heights) # contains the coordinates (x1,y1,x2,y2) of the boundaries of the boxes
```

Figure 4.7: Prediction-making (Own-elaboration)

The next step is to organize the information collected, resize the tensors that we get. That means, that eventually, the coordinates of airplane 0 are in different positions along the different vectors of the different images. So we have to implement a function that guarantees that the airplane 0 in image 0 is actually the same airplane 0 in image 1. The *coordinatestracking* tensor is initialized, and the correct order of the coordinates is ensured by calculating the distance difference between the object of one image to another

with all the objects predicted. This is as easy as making a vector of distances *distance*=[] which stores the distances between objects detected and determines the correct order of the coordinates by choosing the minimum distance (D) from one object to another. For instance:

$$D = \sqrt{[Xcoordinates(currentimage) - Xcoordinates(lastimage)]^2 + [Ycoordinates(currentimage) - Ycoordinates(lastimage)]^2} \quad (4.1)$$

Providing that *D* is the smallest from the corresponding object. Doing this, we achieve that every object is in the correct place on *coordinatestracking* tensor.

One of the first outputs achieved is the detection of the first image of the video, with the objects detected on it, and their corresponding coordinates $[[xy]]$ on display, as seen in Figure 4.8.

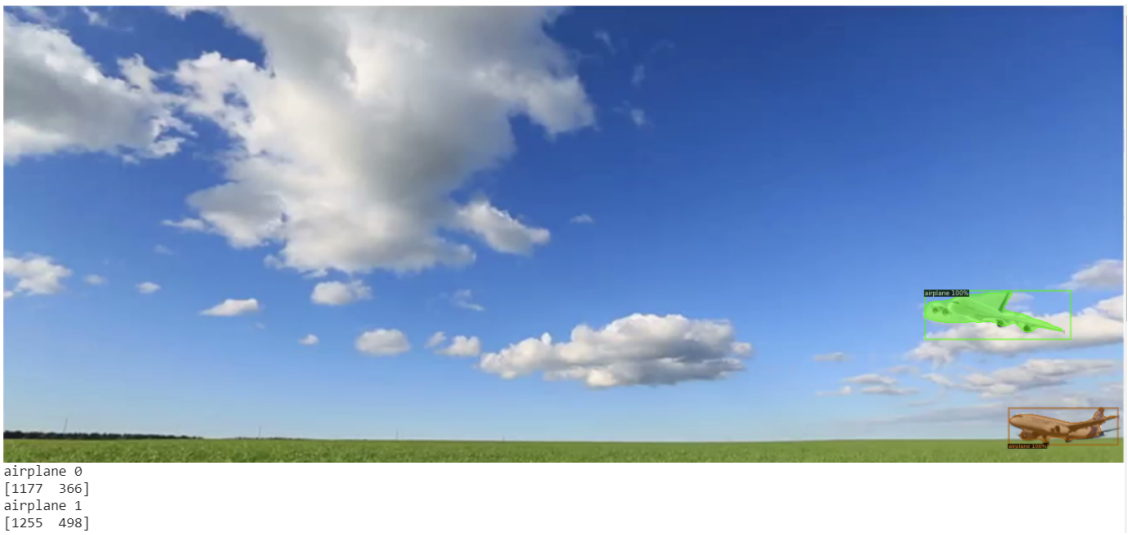


Figure 4.8: Example of detection and coordinates (Own elaboration)

The next section of the algorithm is the treatment of the data, making a motion study of the corresponding objects detected. First, all the tensors and vectors are initialized and the scale is calculated providing the real height of determined COCO objects (airplane, person, car and motorcycle) to find the relation between the pixels of the processed images and real distances. In our case, we have realized it with an airplane:

$$scale = \frac{realheight[airplane]}{boxheight[airplanedetected]} \quad (4.2)$$

Then, we are able to calculate the different parameters of the motion study. We calculate the trajectory using the scale calculated, and afterwards we can calculate iteratively the distances, velocity, and accelerations for each object, as seen in Figure 4.9 .

Finally, after interpolating the graphs we obtain the outputs in Figure 4.10.

If we consider that *airplane0* is the airplane that is taking-off during the video, and *airplane1* the airplane that is taxiing, we can determine that the results obtained are coherent.

```
# iterative vector calculation
i=0
while (len(centers))>i:
  j=0
  while j<(len(centers[0])):
    x[j][i]=scale*coordinatestracking[i][j][0]
    y[j][i]=scale*coordinatestracking[i][j][1]
    j+=1
  i+=1

i=0
while (len(centers)-1)>i:
  j=0
  while j<(len(centers[0])):
    d[j][i]=scale*(math.sqrt((coordinatestracking[i+1][j][0]-coordinatestracking[i][j][0])**2+(coordinatestracking[i+1][j][1]-coordinatestracking[i][j]
    dir[j][i]=(coordinatestracking[i+1][j][1]-coordinatestracking[i][j][1])/(coordinatestracking[i+1][j][0]-coordinatestracking[i][j][0])
    j+=1
  i+=1

i=0
while (len(centers)-2)>i:
  j=0
  while j<(len(centers[0])):
    v[j][i+1]=(d[j][i])/(t[i+1]-t[i])
    j+=1
  i+=1
```

Figure 4.9: Part of the iterative calculation: Trajectory, distance and velocity (Own elaboration)

It can be seen the trajectory of both objects in (a), the total distance traveled in (b), and the velocity (c) and acceleration (d) graphs. It is important to remark that the discontinuities observed are due to the accuracy of the program to detect the coordinates, and also to the continuity and little distance between video frames.

The velocity (c) should be nearly constant in both objects. We can assume that the discontinuities are not so big, and the same happens with the acceleration plot (d). Moreover, the total distance will be larger for the airplane 0 that is executing a take-off maneuver.

Finally, other outputs referring to the maximum velocity, mean velocity, maximum acceleration, and mean acceleration are displayed too (Figure 4.11).

Finally, the last part of the algorithm consists in tracking visually the different objects that appear on the given images. We can see the result in the Figure 4.12:

We can conclude the implementation of object tracking using Detectron2 as a success, we have achieved a coherent and realistic tracking of the situation described.

4.1.2. Difficulties with the implementation

I have considered crucial to dedicate a section to explain the difficulties I had with the implementation of the object tracking part.

Detectron2 made my job harder than I thought at first. First, I faced the following problems:

- Searching a 2D video that contains COCO classes, due to the limitation of Detectron2 of not perceiving the third dimension.
- Seek that the video was recorded with a static, not moving camera. This was due to the simplicity of the implementation.
- Controlling and avoiding complex videos with many objects.
- Detectron2 never detected objects on the same order, appearing and disappering of the detection.

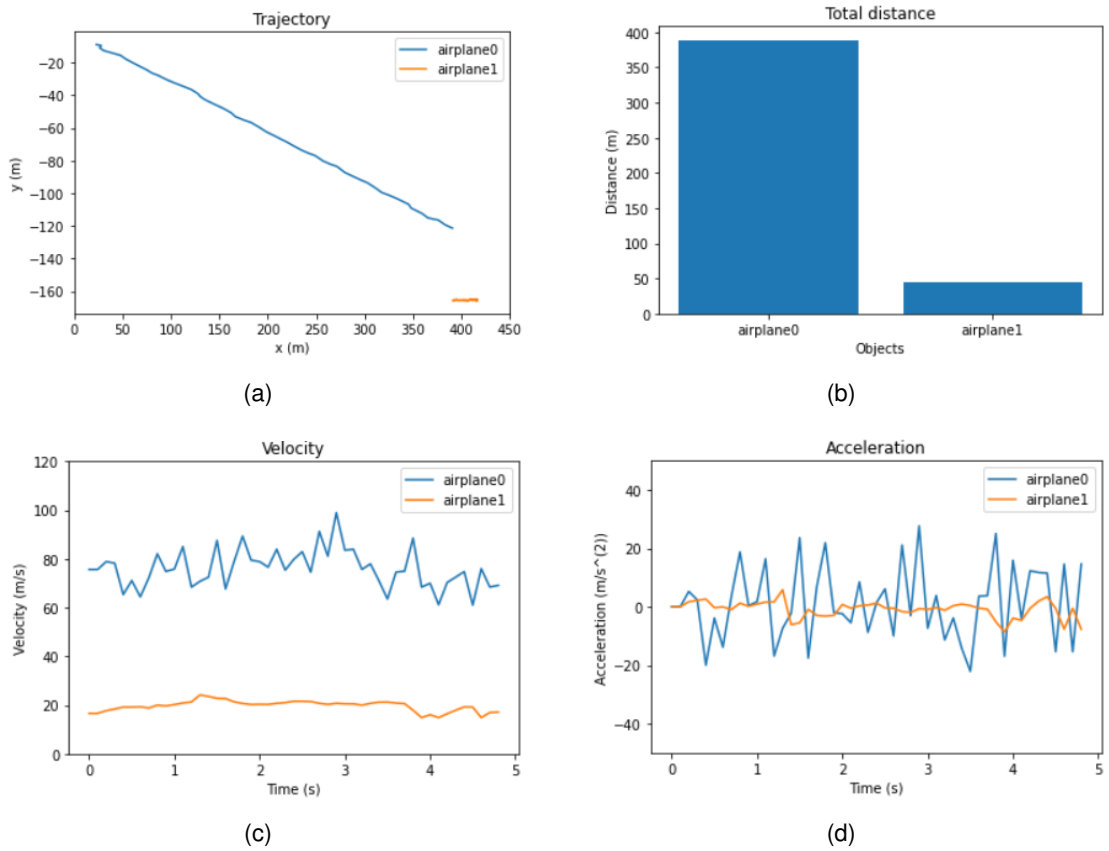


Figure 4.10: Results of motion study (Own elaboration)

```
Maximum velocity of object 0 is 98.9758342738475 m/s.
Mean velocity of object 0 is 75.96597419736023 m/s.
Mean acceleration of object 0 is 1.2043275532408706 m/s.
```

Figure 4.11: Display of mean and maximum values of airplane 0 (Own elaboration)

Figure 4.12: Visual tracking (Own elaboration)

- Looking for a video that objects did not appear and disappear from the point of view, because that would cause problems with the implementation making objects appear

and disappear from the tracking coordinates.

Finally, after many searching that task was solved making a simpler 2D animation on Powerpoint, and the problem dealing with the organization and identification of the object was solved too.

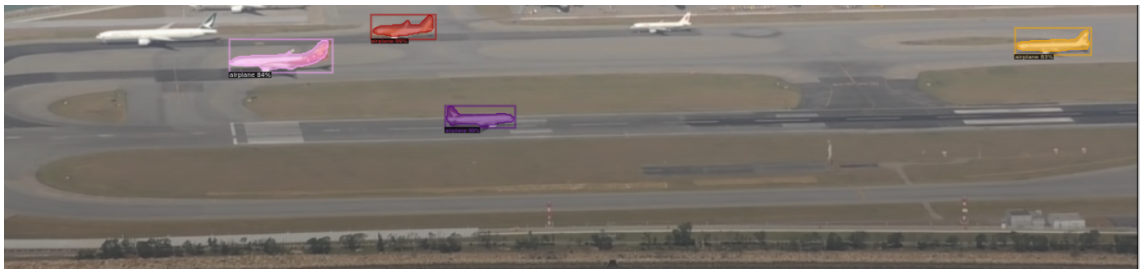


Figure 4.13: Problem with detection (Own elaboration)

As it can be deduced, Detectron2 is not a program to make tracking of objects. Detectron2 is a program that provides object detection on images or videos, and that is the reason why the program detects randomly airplanes in images and other airplanes do not.

So, finally, I succeeded in the final chose and I have achieved the object tracking task proposed at the beginning of the project.

CONCLUSIONS

The purpose of the project was to understand the concept of machine learning, deep learning, and artificial neural networks. Furthermore, it was also crucial the implementation of a series of neural networks, and the implementation of an object tracking method to an advanced pre-trained algorithm, Detectron2. Nowadays, I have to expose that all the objectives have been fulfilled. Along with the project, different decisions have been made to adequate this unknown field to the academic framework, but generally difficulties have been overcomed.

The implementation of different neural networks has given me a more general vision of how machine learning works, because at first machine learning may sound strange and unintelligible. During the implementation of the three neural networks I observed that the architecture of the network was crucial and very important to the result. For example, an algorithm could present overfitting or underfitting in learning curves, and that is a difficult problem to solve. Making good algorithms that fit on data, but also new data, it is a hard job.

Defining parameters of the network is also an important thing to do. Mini-batch size, neurons in each layer, epochs to be made... The field is constantly evolving, and it is important to lean on the experiences of other researchers that have solved similar problems, and adopt their parameters, their architectures, and the techniques that they have used to make a good algorithm.

Referring to the second part, adopting a program with some default parameters and adding new functionalities have been hard for me. Implementing a new code on a complex program implies knowledge of machine learning programming that has been tough to acquire. Anyway, I succeed in my objective with object tracking implementation on Detectron2. A study of the motion of different objects has been made, representing that plots of their trajectory, velocity, acceleration, and total distance traveled by them.

As personal conclusions, I have to admit that I learned of a world that was unknown by me with a lot of new concepts. I have learned to implement different types of neural networks, to identify problems on code and to implement aspects that I have never done before. I have learned to deal with pressure situations and to do a project from zero of a topic not known for me before.

From the programming view, I have to admit that I have learned so much with all these implementations, regarding neural networks and their underline structures. The hardest part has been the last implementation of object tracking where I had various problems on the adaptation with Detectron2, as detailed in section 4.1.2., but finally a simpler solution has been adopted.

I am happy to have learned a lot in such interesting field. As future improvements, I would propose more implementations of even more difficult neural networks, using different databases and making more predictions. Also, for the second part of the project, I would be interesting to implement more functionalities and also to provide the analysis of more features, and more complex videos, making possible the utilization of a video with a non-static camera.

Finally, I have to conclude that this final project has been very productive for me. The

help and patience of my professor Pietro Massignan have been crucial for me, helping me in the most critical moments of work and cheering me from the very start. The machine learning field is a very promising sector with a nice forecast, and I would be happy to work on it in the near future. With the execution of this project, I have discovered an innovative interesting world that I am sure will be present in the lives of all of us.

BIBLIOGRAPHY

[1] Gurney, Kevin. *An introduction to neural networks* [online]. London: Taylor & Francis e-Library, 2004. ISBN 0203451511. Available at: <https://www.inf.ed.ac.uk/teaching/courses/nlu/assets/reading/Gurney_et_al.pdf>. 3

[2] K.S.Jagadish; R.N.Iyengar. *Recent Advances in Structural Engineering*. India: Universities Press (India) Private Limited, 2005. ISBN 8173714932. ix, 4

[3] Ayodele, Taiwo. *Types of Machine Learning Algorithms*. United Kingdom: University of Portsmouth, 2010. ISBN 9789533070346. ix, 7

[4] Moolayil, Jojo. *A Layman's Guide to Deep Neural Networks* [online]. Towards Data Science, 2019. Available at: <https://towardsdatascience.com/a-laymans-guide-to-deep-neural-networks-ddcea24847fb>. ix, 8

[5] Yassin, Gamil. *Build Simple AI .NET Library - Part 3 - Perceptron* [online]. CodeProject, 2017. Available at: <https://www.codeproject.com/Articles/1205732/Build-Simple-AI-NET-Library-Part-Perceptron>. ix, 9

[6] Nielsen, Michael. *Neural Networks and Deep Learning* [online]. Australia: Determination Press, 2015. Available at: <http://neuralnetworksanddeeplearning.com>. ix, 10, 11, 12, 15

[7] Saxena, Sharoon. *Underfitting vs. Overfitting (vs. Best Fitting) in Machine Learning* [online]. Analytics Vidhya, 2020. Available at: <https://www.analyticsvidhya.com/blog/2020/02/underfitting-overfitting-best-fitting-machine-learning/>. ix, 14

[8] CS231n Convolutional Neural Networks for Visual Recognition. *Convolutional Neural Networks (CNNs /ConvNets)* [online]. CS231, 2020. Available at: < https://cs231n.github.io/convolutional-networks/>. ix, 21, 22, 24

[9] Forgus, Rob et al. *Deep Learning Methods for Vision* [online]. New York: NYU, 2012. Available at: <https://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/>. ix, 23

[10] Prabhu, Raghav. *Understading of Convolutional Neural Network (CNN) - Deep Learning* [online]. Medium, 2018. Available at: <https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148>. ix, 23

[11] Budhiraja, Amar. *Dropout in (Deep) Machine Learning* [online]. Medium, 2016. Available at: <https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-mac ix, 25

[12] Ag, Amin. *Batch Normalization in Deep Learning* [online]. Medium, 2019. Available at: <https://medium.com/ai%C2%B3-theory-practice-business/batch-normalization-in-deep-learning-ca215a7a7a5d>. ix, 25

[13] Keras. *Keras* [online]. 2020, Keras. Available at: <https://keras.io/>. ix, 26

[14] Krizhevsky, Alex. *The CIFAR-10 dataset* [online]. California: Alex Krizhevsky, 2009. Available at: <https://www.cs.toronto.edu/~kriz/cifar.html>. ix, 29

[15] Facebook AI. *Detectron2: A PyTorch-based modular object detection library* [online]. Facebook, 2019. Available at: <https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/>. ix, 35

[16] Colab. *Detectron2 Beginner's Tutorial* [online]. Facebook, 2020. Available at: <https://colab.research.google.com/drive/16jcaJoc6bCFAQ96jDe2HwtXj7BMD_-m5>. ix, 36

[17] Cornell University. *Microsoft COCO: Common Objects in Context* [online]. Cornell University, 2014. Available at: <https://cocodataset.org/>. ix, 37

[18] Salamanca University. *Redes Neuronales* [online]. Available at: <http://avellano.fis.usal.es/~lalonso/RNA/index.htm>.

[19] The University of Queensland. *What is a neuron?* [online]. Australia: The University of Queensland, 2019. Available at: <http://qbi.uq.edu.au/brain/brain-anatomy/what-neuron>.

[20] HealthLine. *What are neurons?* [online]. New York: Healthline, 2020. Available at: <https://www.healthline.com/health>.

[21] Wikipedia. *Computer Science* [online]. Wikimedia Foundation, 2020. Available at: <https://en.wikipedia.org/wiki/Computer_science>.

[22] Wikipedia. *Artificial Intelligence* [online]. Wikimedia Foundation, 2020. Available at: <https://en.wikipedia.org/wiki/Artificial_intelligence>.

[23] Mathworks. *Machine Learning: Tres cosas que es necesario saber* [online]. The Mathworks, Inc. 2020. Available at: <https://es.mathworks.com/discovery/machine-learning.html>.

[24] Simon, Phil. *Too Bit To Ignore: The Business Case for Big Data*. Hoboken, New Jersey: John Wiley & Sons, Inc. 2013. ISBN 9781119217848. 5

[25] Mitchell, Tom. *Machine Learning*. McGraw Hill Science/Engineering/Math, 1997. ISBN 9780070428072.

[26] Wikipedia. *Timeline of machine learning* [online]. Wikimedia Foundation, 2020. Available at: <https://en.wikipedia.org/wiki/Timeline_of_machine_learning>.

[27] Internet Society. *Artificial Intelligence and Machine Learning: Policy Paper* [online]. Reston: Internet Society, 2020. Available at: <https://www.internetsociety.org/resources/doc/2017/artificial-intelligence-and-machine-learning-policy-paper/?gclid=CjwKCAjw-YT1BRAFEiwAd2WRtqv_J1oMlJ9_ZrCflgi_xArMgtCUNTumFJBPm-NoR4dQuv69_sTQdRoCoeIQAvD_BwE>.

[28] Mohammed, Mohssen et al. *Machine Learning. Algorithms an Applications*. Boca Raton: Taylor & Francis Group, LLC, 2017. ISBN 9781498705387.

[29] Hurwitz, Judith; Kirsch, Daniel. *Machine Learning for dummies* [online]. Hoboken: John Wiley & Sons, Inc, 2018. ISBN 978119454946. Available at: <https://www.ibm.com/downloads/cas/GB8ZMQZ3>.

[30] Geeks for geeks. *Semi-Supervised Learning* [online]. Noida: Geeks for Geeks, 2019. Available at: <https://www.geeksforgeeks.org/ml-semi-supervised-learning/>.

[31] Brownlee, Jason. *What is Deep Learning?* [online]. Australia: Machine Learning Mastery, 2019. Available at: <https://machinelearningmastery.com/what-is-deep-learning/>.

[32] Brownlee, Jason. *Overfitting and Underfitting With Machine Learning Algorithms* [online]. Australia: Machine Learning Mastery, 2016. Available at: <https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/>.

[33] Jupyter Notebook. *Jupyter Notebook* [online]. Project Jupyter, 2020. Available at: <https://jupyter.org/>.

[34] Brownlee, Jason. *How to use Learning Curves to Diagnose Machine Learning Model Performance* [online]. Australia: Machine Learning Mastery, 2019. Available at: <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>.

APPENDICES

APPENDIX A. MNIST DEEP NEURAL NETWORK

The appendix *"MNIST Deep Neural Network (DNN)"* shows the implementation of a Deep Neural Network based on the MNIST database. It is written in Python 3.8 language on a Jupyter Notebook environment. In this appendix, the full code and training process can be found.

network

July 5, 2020

1 MNIST DNN

Adapted from "Neural Network and Deep Learning"- Michael Nielsen (Dec 2019)

[3]:
```
class Network(object):

    def __init__(self, sizes):

        """The list ''sizes'' contains the number of neurons in the
        respective layers of the network. The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1."""

        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Implements the sigmoid function"""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent (SGD).  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out."""

        training_data = list(training_data)
        n = len(training_data)

        if test_data:
```

```
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.
↪evaluate(test_data),n_test));
            else:
                print("Epoch {} complete".format(j))

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
```

```
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        """Return the number of test inputs for which the neural
        network outputs the correct result."""
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

2 Train the model

[4]:
```
import mnist_loader
```

[5]:
```
training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

[6]:
```
import network
```

[7]:
```
net = network.Network([784, 30, 10]) # A network with first layer (784␣
↪neurons),30 hidden neurons and 10 neurons in the third layer.
```

[8]:
```
network=net.SGD(training_data, 30, 10, 3.0, test_data=test_data) # Learn with␣
↪30 epochs, with a mini batch size of 10, and a learning rate 3.0.
```

```
Epoch 0 : 8284 / 10000
Epoch 1 : 8353 / 10000
Epoch 2 : 8452 / 10000
Epoch 3 : 8460 / 10000
Epoch 4 : 9269 / 10000
Epoch 5 : 9376 / 10000
Epoch 6 : 9433 / 10000
Epoch 7 : 9406 / 10000
Epoch 8 : 9413 / 10000
Epoch 9 : 9407 / 10000
Epoch 10 : 9444 / 10000
Epoch 11 : 9426 / 10000
Epoch 12 : 9451 / 10000
Epoch 13 : 9454 / 10000
Epoch 14 : 9429 / 10000
Epoch 15 : 9459 / 10000
Epoch 16 : 9468 / 10000
Epoch 17 : 9458 / 10000
Epoch 18 : 9470 / 10000
Epoch 19 : 9496 / 10000
Epoch 20 : 9472 / 10000
Epoch 21 : 9436 / 10000
Epoch 22 : 9490 / 10000
Epoch 23 : 9477 / 10000
Epoch 24 : 9483 / 10000
Epoch 25 : 9483 / 10000
Epoch 26 : 9478 / 10000
Epoch 27 : 9502 / 10000
Epoch 28 : 9507 / 10000
Epoch 29 : 9498 / 10000
```

APPENDIX B. MNIST CONVOLUTIONAL NEURAL NETWORK USING KERAS

The appendix *"MNIST Convolutional Neural Network (CNN) using Keras"* shows the implementation of a Convolutional Neural Network based on the MNIST database. It is written in Python 3.8 language on a Jupyter Notebook environment using the Keras API. In this appendix, the full code and training process can be found. Also, graphs about training and validation accuracy/loss are attached.

MNIST CNN using Keras

July 7, 2020

1 MNIST Deep Convolutional Neural Network (CNN) using Keras

Adapted from https://keras.io/examples/mnist_cnn/

2 IMPORT DEPENDENCIES

2.0.1 First is needed to import all modules and dependencies to make CNN work:

[15]:
```
from __future__ import print_function
import keras # Import keras deep learning library.
from keras.datasets import mnist # Import the MNIST dataset from keras.
from keras.models import Sequential # The simplest and more friendly model to␣
 ↪program keras and define the layer's structure.
from keras.layers import Dense, Dropout, Flatten # Import the type of layers␣
 ↪that will be used from keras.
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.preprocessing.image import load_img, img_to_array # To test the CNN␣
 ↪and rescale an entering image.
import os # For looking and directories and import settings.
import matplotlib.pyplot as plt
import timeit # We import it to calculate how many time lasts our models to be␣
 ↪trained.
```

3 DEFINE NETWORK PARAMETERS

3.0.1 The next lines represent the setting of the parameters of the Convolutional Neural Network:

[16]:
```
batch_size = 128
num_classes = 10 # The 10 digits 1,2,3,4,5,6... collected in 10 clases.
epochs = 12
```

4 SAVE THE MODEL

[17]:
```
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'mnist_cnn_trained.h5'
```

5 LOAD DATASET

[18]:
```
# Input image dimensions.
img_rows, img_cols = 28, 28

# Load dataset.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Prepare pixel data.
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

#Summarize loaded dataset
print('Loaded dataset dimensions:')
print('-Train: Xtraining=%s, Ytraining=%s' % (x_train.shape, y_train.shape))
print('-Test: Xtraining=%s, Ytraining=%s' % (x_test.shape, y_test.shape))
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
Loaded dataset dimensions:
-Train: Xtraining=(60000, 28, 28, 1), Ytraining=(60000,)
-Test: Xtraining=(10000, 28, 28, 1), Ytraining=(10000,)
60000 train samples
10000 test samples
```

In the following lines, the structure of the CNN is described. With the command model.add() all the desired layers can be added to the entire neural network.

5.1 BUILD AND TRAIN THE CNN

5.2 (this whole section may be skipped if the CNN has already been trained)

[19]:
```
model = Sequential() #We import the sequential model of keras.

#Next we describe the arquitechture of the CNN (layers).
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

5.2.1 Use an optimizer to train the model

[20]:
```
# Use Adadelta optimizer.
opt = keras.optimizers.Adadelta(learning_rate=1.0, rho=0.95)
#Normally this terms are by default.

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

5.2.2 Train the Convolutional Neural Network:

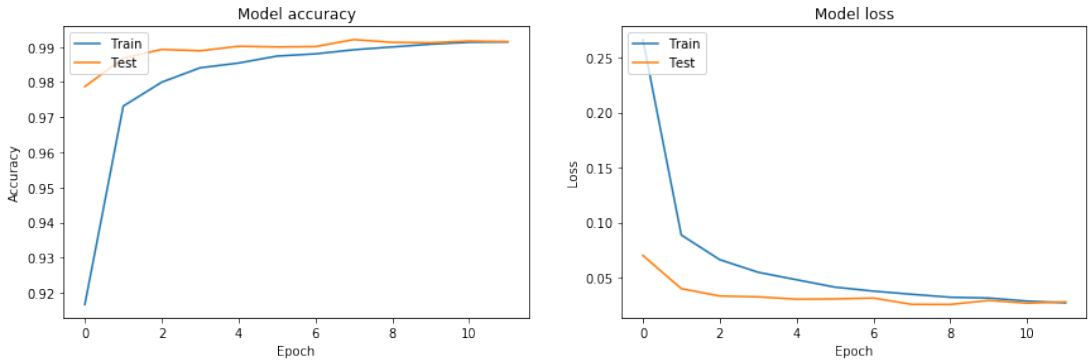
[21]:
```
start = timeit.default_timer()

history= model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

stop = timeit.default_timer()


print('Time: ', stop - start)

# Save model and weights.
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
```

```
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model.
result = model.evaluate(x_test, y_test, verbose=0)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 213s 4ms/step - loss: 0.2665 -
accuracy: 0.9167 - val_loss: 0.0703 - val_accuracy: 0.9787
Epoch 2/12
60000/60000 [==============================] - 100s 2ms/step - loss: 0.0888 -
accuracy: 0.9732 - val_loss: 0.0401 - val_accuracy: 0.9868
Epoch 3/12
60000/60000 [==============================] - 99s 2ms/step - loss: 0.0664 -
accuracy: 0.9800 - val_loss: 0.0334 - val_accuracy: 0.9893
Epoch 4/12
60000/60000 [==============================] - 100s 2ms/step - loss: 0.0549 -
accuracy: 0.9841 - val_loss: 0.0326 - val_accuracy: 0.9889
Epoch 5/12
60000/60000 [==============================] - 99s 2ms/step - loss: 0.0482 -
accuracy: 0.9854 - val_loss: 0.0305 - val_accuracy: 0.9902
Epoch 6/12
60000/60000 [==============================] - 98s 2ms/step - loss: 0.0414 -
accuracy: 0.9874 - val_loss: 0.0307 - val_accuracy: 0.9900
Epoch 7/12
60000/60000 [==============================] - 98s 2ms/step - loss: 0.0378 -
accuracy: 0.9880 - val_loss: 0.0315 - val_accuracy: 0.9901
Epoch 8/12
60000/60000 [==============================] - 99s 2ms/step - loss: 0.0349 -
accuracy: 0.9892 - val_loss: 0.0258 - val_accuracy: 0.9921
Epoch 9/12
60000/60000 [==============================] - 99s 2ms/step - loss: 0.0322 -
accuracy: 0.9900 - val_loss: 0.0257 - val_accuracy: 0.9913
Epoch 10/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0315 -
accuracy: 0.9908 - val_loss: 0.0294 - val_accuracy: 0.9912
Epoch 11/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0287 -
accuracy: 0.9913 - val_loss: 0.0269 - val_accuracy: 0.9917
Epoch 12/12
60000/60000 [==============================] - 101s 2ms/step - loss: 0.0272 -
accuracy: 0.9915 - val_loss: 0.0281 - val_accuracy: 0.9915
Time:  1316.3387901000096
Saved trained model at C:\Users\claud\OneDrive\Escritorio\TFG\TFG\JupyterNoteboo
k\saved_models\mnist_cnn_trained.h5
```

6 PLOT TRAINING AND VALIDATION ACCURACY/LOSS VALUES

[22]:
```
# Accuracy
plt.subplot(121)
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')

# Loss
plt.subplot(122)
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.subplots_adjust(bottom=0.1, right=2, top=0.9)
plt.show()

# Final classification accuracy
FinalAccuracy=result[1]*100
FinalLoss=result[0]
print('Test accuracy is'+' '+ str(FinalAccuracy)+' '+'%'+'.')
print('Test loss is'+' '+str(FinalLoss)+'.')
```




```
Test accuracy is 99.15000200271606 %.
Test loss is 0.02810708877939178.
```

APPENDIX C. CIFAR-10 CONVOLUTIONAL NEURAL NETWORK USING KERAS

The appendix *"CIFAR-10 Convolutional Neural Network (CNN) using Keras"* shows the implementation of a Convolutional Neural Network based on the CIFAR-10 dataset. It is written in Python 3.8 language on a Jupyter Notebook environment using the Keras API. In this appendix, the full code and training process can be found.

CIFAR-10 CNN using Keras

August 23, 2020

1 CIFAR-10 Deep Convolutional Neural Network (CNN) using Keras

Adapted from https://keras.io/examples/cifar10_cnn/

2 IMPORT DEPENDENCIES

2.0.1 First is needed to import all modules and dependencies to make CNN work:

[ ]:
```
from __future__ import print_function
import keras # Import keras deep learning library.
from keras.datasets import cifar10 # Import the CIFAR-10 dataset from keras.
from keras.models import Sequential # The simplest and more friendly model to␣
↪program keras and define the layer's structure.
from keras.layers import Dense, Dropout, Activation, Flatten,␣
↪BatchNormalization # Import the type of layers that will be used from keras.
from keras.layers import Conv2D, MaxPooling2D # More layers to be imported.
from keras.optimizers import SGD, RMSprop # Import the optimizer.
from keras.models import load_model # To load a trained model.
from keras.preprocessing.image import load_img, img_to_array # To test the CNN␣
↪and rescale an entering image.
import os # For looking and directories and import settings.
import matplotlib.pyplot as plt
import timeit # We import it to calculate how many time lasts our models to be␣
↪trained.
```

3 DEFINE NETWORK PARAMETERS

3.0.1 The next lines represent the setting of the parameters of the Convolutional Neural Network:

[ ]:
```
batch_size = 32
num_classes = 10 # CIFAR-10: 10 classes.
epochs = 10 # Times the network will be trained with batches method.
num_predictions = 20
rmsprop= True # Wether or not use rmsprop optimizer.
dropout= True # Wether or not use dropout regularization technique.
```

```
batchnormalization= False # Wether or not use batch normalization technique.
```

4 SAVE THE MODEL

[ ]:
```
if SGD:
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_initialmodel.h5'

if rmsprop and not(dropout,batchnormalization):
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_rmspropmodel.h5'

if rmsprop and dropout and not(batchnormalization):
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_dropoutmodel.h5'

if rmsprop and not(dropout) and (batchnormalization):
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_batchnormalizationmodel.h5'

if rmsprop and dropout and batchnormalization:
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_finalmodel.h5'

else:
    save_dir = os.path.join(os.getcwd(), 'saved_models')
    model_name = 'keras_trained_newmodel.h5'
```

5 LOAD DATASET

[ ]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data() # Importation of␣
↪test and training data.

#Summarize loaded dataset.
print('Loaded dataset dimensions:')
print('-Train: Xtraining=%s, Ytraining=%s' % (x_train.shape, y_train.shape))
print('-Test: Xtraining=%s, Ytraining=%s' % (x_test.shape, y_test.shape))
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

5.1 BUILD AND TRAIN THE CNN

5.1.1 (this whole section may be skipped if the CNN has already been trained)

[ ]:
```
model = Sequential() #We import the sequential model of keras.

#Next we describe the arquitechture of the CNN (layers).
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
if dropout:
    model.add(Dropout(0.25))


model.add(Conv2D(64, (3, 3), padding='same'))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
if dropout:
    model.add(Dropout(0.25))


model.add(Flatten())
model.add(Dense(512))
if batchnormalization:
    model.add(BatchNormalization())
model.add(Activation('relu'))
if dropout:
    model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

5.1.2 Use an optimizer to train the model

[ ]:
```
if rmsprop:
    opt = keras.optimizers.RMSprop(learning_rate=0.0001, decay=1e-6)
#Normally the parameters of optimizers are by default.
else:
    opt = keras.optimizers.SGD(learning_rate=0.01)

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

#Prepare pixel data.
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

5.1.3 Train the Convolutional Neural Network:

[ ]:
```
start = timeit.default_timer()

history=model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)

stop = timeit.default_timer()

print('Time: ', stop - start)

# Save model and weights.
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model.
result = model.evaluate(x_test, y_test, verbose=1)
```

6 PLOT TRAINING AND VALIDATION ACCURACY/LOSS VALUES

[ ]:
```
# Accuracy
plt.subplot(121)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.ylim((0,1))
plt.legend(['Train', 'Test'], loc='upper left')

# Loss
plt.subplot(122)
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.ylim((0,2.5))
plt.subplots_adjust(bottom=0.1, right=2, top=0.9)
plt.show()

# Final classification accuracy
FinalAccuracy=result[1]*100
FinalLoss=result[0]
print('Test accuracy is'+' '+ str(FinalAccuracy)+' '+'%'+'.')
print('Test loss is'+' '+str(FinalLoss)+'.')
```

7 TEST THE CNN

[ ]:
```
# Import the trained model.
model_path = os.path.join(save_dir, model_name)
model = load_model(model_path)

#Define the label names of the outputs of the trained model. In this case the␣
 ↪CIFAR-10 model labels.
def load_label_names():
    return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',␣
 ↪'horse', 'ship', 'truck']

def classify(sample_image):
    # Load the image
    img = load_img(sample_image, target_size=(32, 32))
```

```
    # Convert to array
    img = img_to_array(img)
    # Reshape into a single sample with 3 channels
    img = img.reshape(1, 32, 32, 3)
    # Prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    result = model.predict_classes(img)
    print(load_label_names()[result[0]])
```

7.0.1 Make a prediction:

[18]:
```
# It can be any dimensions input image. The function "classify"
#reshape it and make the prediction.
classify('airplane1.jpg')
```

```
airplane
```

APPENDIX D. COMPARISON OF THE RESULTS OBTAINED BY CIFAR-10 CNN CODE USING KERAS

The appendix *"Comparison of the results obtained by CIFAR-10 CNN code using Keras"* shows the results obtained from the implementation of a Convolutional Neural Network based on the CIFAR-10 dataset. It is written in Python 3.8 language on a Jupyter Notebook environment using the Keras API. In this appendix, graphs about training and validation accuracy/loss of five experimental cases are attached.

Experimental cases attained:

1. Simple model using stochastic gradient descent optimizer (SGD)
2. Simple model using rmsprop optimizer
3. Model using rmsprop optimizer and dropout regularization technique
4. Model using rmsprop optimizer and batch normalization technique
5. Model using rmsprop optimizer and both regularization techniques (dropout and batch normalization)

CIFAR-10 CNN COMPARISON OF THE RESULTS

April 9, 2020

1 COMPARISON OF THE RESULTS

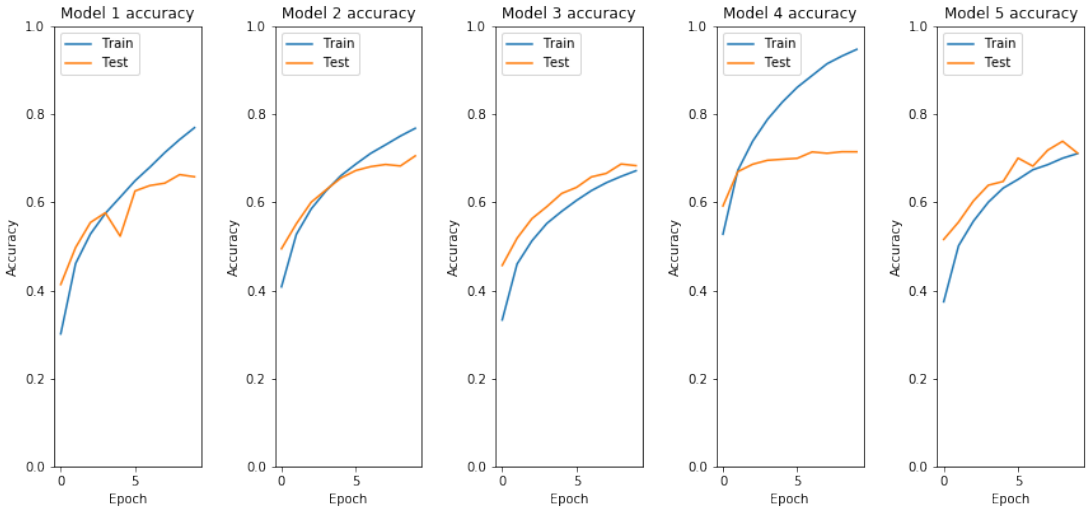
1.0.1 This comparison is made between five trained models:

1. Simple model using stochastic gradient descent optimizer (SGD).
2. Simple model using rmsprop optimizer.
3. Model using rmsprop optimizer and dropout regularization technique.
4. Model using rmsprop optimizer and batch normalization regularization technique.
5. Model using rmsprop optimizer and both regularization techniques (dropout and batch normali

1.1 Accuracy

[205]:
```
plt.subplot(151)
plt.plot(accuracy1)
plt.plot(accuracyref1)
plt.ylim((0,1))
plt.title('Model 1 accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')


plt.subplot(152)
plt.plot(accuracy2)
plt.plot(accuracyref2)
plt.ylim((0,1))
plt.title('Model 2 accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(153)
plt.plot(accuracy3)
plt.plot(accuracyref3)
plt.ylim((0,1))
plt.title('Model 3 accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
```

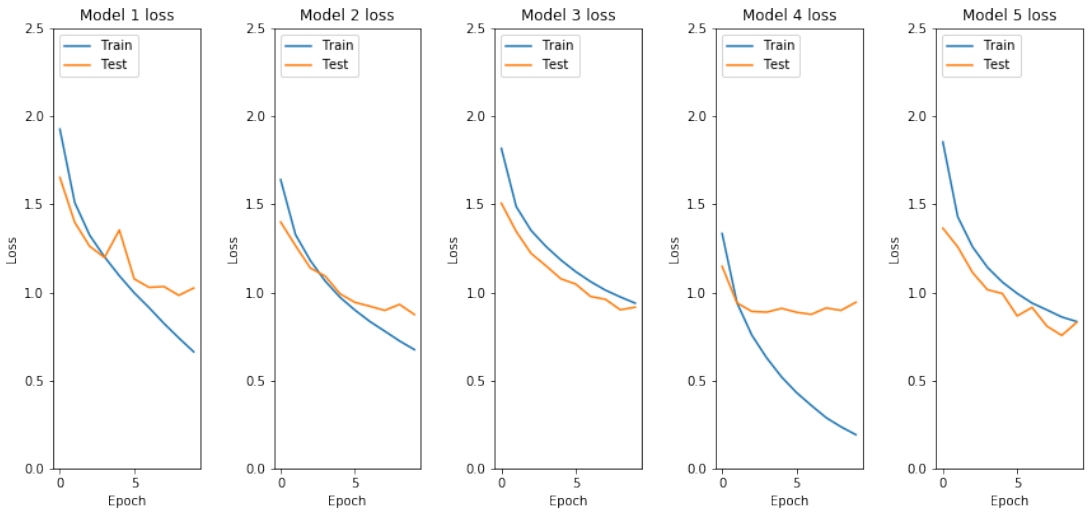
```
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(154)
plt.plot(accuracy4)
plt.plot(accuracyref4)
plt.ylim((0,1))
plt.title('Model 4 accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(155)
plt.plot(accuracy5)
plt.plot(accuracyref5)
plt.ylim((0,1))
plt.title('Model 5 accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.subplots_adjust(bottom=0.1, right=2, top=1.3, wspace=0.5, hspace=0.5)
plt.show()
```



1.2 Loss

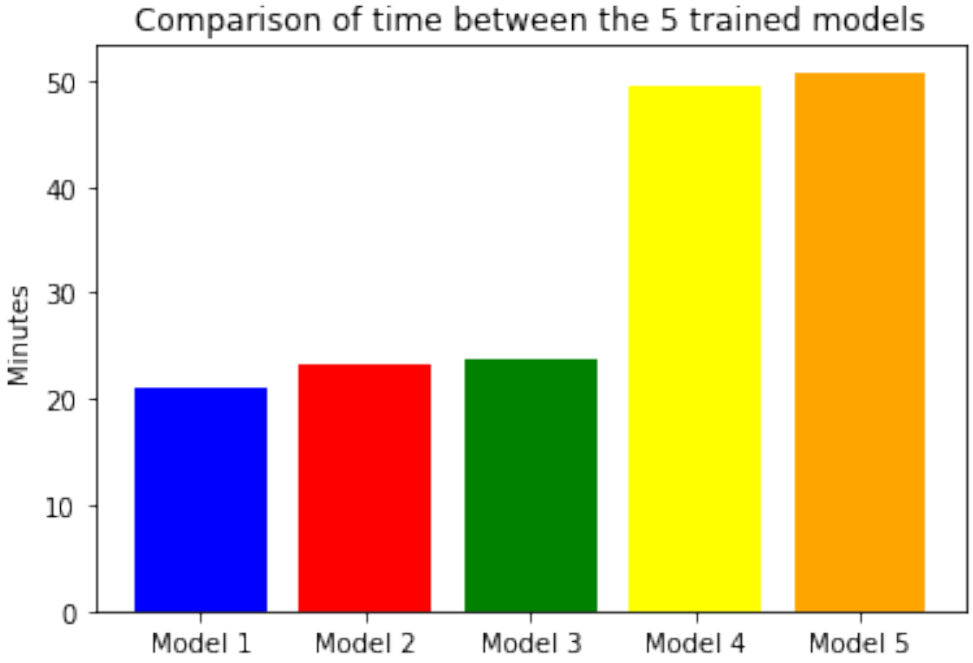
[206]:
```
plt.subplot(151)
plt.plot(loss1)
plt.plot(lossref1)
plt.ylim((0,2.50))
plt.title('Model 1 loss')
```

```
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')


plt.subplot(152)
plt.plot(loss2)
plt.plot(lossref2)
plt.ylim((0,2.50))
plt.title('Model 2 loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(153)
plt.plot(loss3)
plt.plot(lossref3)
plt.ylim((0,2.50))
plt.title('Model 3 loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(154)
plt.plot(loss4)
plt.plot(lossref4)
plt.ylim((0,2.50))
plt.title('Model 4 loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.subplot(155)
plt.plot(loss5)
plt.plot(lossref5)
plt.ylim((0,2.50))
plt.title('Model 5 loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.subplots_adjust(bottom=0.1, right=2, top=1.3, wspace=0.5, hspace=0.5)
plt.show()
```



1.3 Time

[207]:
```
time=[]
x=['Model 1','Model 2','Model 3', 'Model 4', 'Model 5']
time.append(float(time1)/60)
time.append(float(time2)/60)
time.append(float(time3)/60)
time.append(float(time4)/60)
time.append(float(time5)/60)
plt.bar(x,time,color=['blue','red','green','yellow','orange'])
plt.title('Comparison of time between the 5 trained models')
plt.ylabel('Minutes')
plt.show()
```


Comparison of time between the 5 trained models
Minutes
0
10
20
30
40
50
Model 1
Model 2
Model 3
Model 4
Model 5

APPENDIX E. OBJECT TRACKING

The appendix *"Object Tracking"* shows the implementation of object tracking into the Detectron2 program provided by Facebook. It is written in Python 3.8 language on the Google Colab environment. In this appendix, full code is presented.

Object Tracking based on Detectron2

Adapted from https://colab.research.google.com/drive/16jcaJoc6bCFAQ96jDe2HwtXj7BMD_-m5#scrollTo=YU5_W8wJF02F.

Here, we will go through some basics usage of detectron2, including the following:

- Run inference on images or videos, with an existing detectron2 model
- Train a detectron2 model on a new dataset

Then, we will go to my implementation, Object Tracking.

- Tracking of objects

Install detectron2

```
# install dependencies:
!pip install pyyaml==5.1 pycocotools>=2.0.1
import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version
# opencv is pre-installed on colab
```

```
# install detectron2: (Colab has CUDA 10.1 + torch 1.6)
# See https://detectron2.readthedocs.io/tutorials/install.html for instructions
assert torch.__version__.startswith("1.6")
!pip install detectron2 -f https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.6
```

```
# Some basic setup:
# Setup detectron2 logger
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import os, json, cv2, random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
```

```
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
```

Run a pre-trained detectron2 model

```
from google.colab import drive
drive.mount('/content/drive')
```

```
# import the images from Google Drive
i=0
images=[]
for i in range(50):
    filename = str(i) + ".PNG"
    im=cv2.imread("/content/drive/My Drive/"+filename)
    images.append(im)
```

Then, we create a detectron2 config and a detectron2 `DefaultPredictor` to run inference on this image.

```
cfg = get_cfg()
# add project-specific config (e.g., TensorMask) here if you're not running a model in det
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FP
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5  # set threshold for this model
# Find a model from detectron2's model zoo. You can use the https://dl.fbaipublicfiles...
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50
predictor = DefaultPredictor(cfg)
outputs=[]
for i in range(len(images)):
    outputs.append(predictor(images[i]))
```

```
#outputs of learning
import tensorflow as tf
import math

#get instances and boxes predictions
centers=[]
heights=[]
for i in range(len(images)):
   outputs[i]["instances"].pred_classes
   outputs[i]["instances"].pred_boxes
   heights.append(outputs[i]["instances"].pred_boxes)
   heights[i]=np.array(heights[i])
   centers.append(outputs[i]["instances"].pred_boxes.get_centers())
   centers[i]=centers[i].cpu()
   centers[i]=centers[i].numpy()
   centers[i]=centers[i].astype(int)
   centers[i]=np.array(centers[i])

centers=np.array(centers) # contains the coordinates (x,y) of the centers of the boxes
```

```
centers=np.array(centers) # contains the coordinates (x,y) of the centers of the boxes
heights=np.array(heights) # contains the coordinates (x1,y1,x2,y2) of the boundaries of th

#COCO classes dataset
classes=["person","bycicle","car","motorcycle","airplane","bus","train","truck","boat","tr

#draw of the first image predictions
v = Visualizer(images[0][:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=1.
v = v.draw_instance_predictions(outputs[0]["instances"].to("cpu"))
cv2_imshow(v.get_image()[:, :, ::-1])

# recoginize the classes detected
hf=outputs[0]["instances"].pred_classes.cpu()
i=0
while i<len(hf):
  print(classes[hf[i]]+" "+str(i))
  print(centers[0][i])
  i+=1

# controlling of the shape of the vectors
maxj=0
i=0
j=[]
while i<len(centers):
  j.append(len(centers[i]))
  i+=1
positionmin=j.index(min(j))

#initialization of coordinatestracking vector
coordinatestracking=tf.zeros([len(centers), len(centers[0]), 2])
coordinatestracking=coordinatestracking.cpu()
coordinatestracking=coordinatestracking.numpy()
coordinatestracking=np.array(coordinatestracking)
coordinatestracking=coordinatestracking.astype(int)
coordinatestracking[0]=centers[0]
size=len(centers)

#check that objects occupy the right position (airplane 0 in 0 position, airplane 1 in 1 p
i=0
while i<(len(centers)-1):
  w=0
  while w<(len(centers[0])):
    j=0
    distance=[]
    while j<(len(centers[0])):
      distance.append(math.sqrt((centers[i+1][j][0]-coordinatestracking[i][w][0])**2+(cent
      j+=1
    positionmind=distance.index(min(distance))
    coordinatestracking[i+1][w]=(centers[i+1][positionmind][0],centers[i+1][positionmind][
    w+=1
  i+=1

heights=heights[0] # we only care about the initial box boundaries
print("Heights:"+" "+str(heights))
print("This is the coordinates vector:"+" "+ str(coordinatestracking))
```

```
#Calculation of motion
import matplotlib.pyplot as plt
import math

#Vector initializing
x=tf.zeros([len(centers[0]), len(centers), 1])
x=x.cpu()
x=x.numpy()
x=np.array(x)
x=x.astype(float)

y=tf.zeros([len(centers[0]), len(centers), 1])
y=y.cpu()
y=y.numpy()
y=np.array(y)
y=y.astype(float)

d=tf.zeros([len(centers[0]), len(centers)-1, 1])
d=d.cpu()
d=d.numpy()
d=np.array(d)
d=d.astype(float)

v=tf.zeros([len(centers[0]), len(centers)-1, 1])
v=v.cpu()
v=v.numpy()
v=np.array(v)
v=v.astype(float)

a=tf.zeros([len(centers[0]), len(centers)-1, 1])
a=a.cpu()
a=a.numpy()
a=np.array(a)
a=a.astype(float)

dir=tf.zeros([len(centers[0]), len(centers)-1, 1])
dir=dir.cpu()
dir=dir.numpy()
dir=np.array(dir)
dir=dir.astype(float)

vmax=[]
vmax=np.zeros((len(centers[0]),))

vmean=[]
vmean=np.zeros((len(centers[0]),))

amax=[]
amax=np.zeros((len(centers[0]),))

amean=[]
amean=np.zeros((len(centers[0]),))

totald=[]
```

```
totald=np.zeros((len(centers[0]),))

directionbetweeninitialandfinal=[]
directionbetweeninitialandfinal=np.zeros((len(centers[0])-1,))

# scale calculation
classesscale=[0,2,3,4] #"person","car","motorcycle","airplane"
heightsscale=[1.75,1.45,1.20,15.00] #alturas en la realidad: [0]persona [1]coche [2]moto [
classes=outputs[0]["instances"].pred_classes.cpu()

j=0
while j<len(classesscale):
  i=0
  if (classes[i]==classesscale[j]):
    scale=heightsscale[i]/(0.007*(np.array(abs(heights[0][1].cpu()-heights[0][0].cpu()))))
    i+=1
  j+=1

# time vector calculation
t=[0]
j=0
while j<len(centers-1):
  t.append(t[j]+0.1)
  j+=1

xaxis=[0]
j=0
while j<(len(centers)-2):
  xaxis.append(t[j]+0.1)
  j+=1

# iterative vector calculation
i=0
while (len(centers))>i:
  j=0
  while j<(len(centers[0])):
    x[j][i]=scale*coordinatestracking[i][j][0]
    y[j][i]=scale*coordinatestracking[i][j][1]
    j+=1
  i+=1

i=0
while (len(centers)-1)>i:
  j=0
  while j<(len(centers[0])):
    d[j][i]=scale*(math.sqrt((coordinatestracking[i+1][j][0]-coordinatestracking[i][j][0])
    j+=1
  i+=1

#set initial values of velocity
v[0][0]=(d[0][0])/(t[1]-t[0])
v[1][0]=(d[1][0])/(t[1]-t[0])

#velocity calculation
i=0
```

```
while (len(centers)-2)>i:
  j=0
  while j<(len(centers[0])):
    v[j][i+1]=(d[j][i])/(t[i+1]-t[i])
    j+=1
  i+=1

j=0
while j<len(centers[0]):
  totald[j]=(sum(d[j]))
  vmax[j]=(max(v[j]))
  vmean[j]=(sum(v[j])/len(v[j]))
  j+=1

vmeantotal=(sum(vmean)/len(centers[0]))

#velocity interpolation
j=0
while j<(len(centers[0])):
  i=0
  while i<(len(centers)-3):
    resta=abs(v[j][i+1]-v[j][i])
    if (resta)>(0.25*v[j][i]):
      v[j][i+1]=(v[j][i]+vmeantotal+v[j][i+1]+vmean[j])/4
      v[j][i+1]=(v[j][i+1]+v[j][i])/2
    i+=1
  j+=1

j=0
while j<(len(centers[0])):
  i=len(centers)-3
  while i<=(len(centers)-2):
    resta=abs(v[j][i]-v[j][i-1])
    if (resta)>(0.25*v[j][i]):
      v[j][i]=(v[j][i-1]+0.01*v[j][i-1])
    i+=1
  j+=1

#set initial values of acceleration
a[0][0]=abs((v[0][1]-v[0][0]))/(t[1]-t[0])
a[1][0]=abs((v[1][1]-v[1][0]))/(t[1]-t[0])

#acceleration calculation
i=0
while (len(centers)-2)>i:
  j=0
  while j<(len(centers[0])):
    a[j][i+1]=(v[j][i+1]-v[j][i])/(t[i+1]-t[i])
    j+=1
  i+=1

j=0
while j<len(centers[0]):
  totald[j]=(sum(d[j]))
  vmax[j]=(max(v[j]))
```

```
    vmean[j]=(sum(v[j])/len(v[j]))
    amax[j]=(max(a[j]))
    amean[j]=(sum(a[j])/len(a[j]))
    j+=1

  ameantotal=(sum(amean)/len(centers[0]))

  #acceleration interpolation
  j=0
  while j<(len(centers[0])):
    i=0
    while i<(len(centers)-3):
      resta=abs(a[j][i+1])-abs(a[j][i])
      if (resta)>(0.05*a[j][i]):
        a[j][i+1]=(a[j][i]+ameantotal+a[j][i+1])/3
        a[j][i+1]=(a[j][i+1]+a[j][i])/2
      i+=1
    j+=1

  j=0
  while j<(len(centers[0])):
    i=len(centers)-3
    while i<=(len(centers)-2):
      resta=abs(a[j][i]-a[j][i-1])
      if (resta)>(0.05*a[j][i]):
        a[j][i]=(a[j][i-2]+0.001*a[j][i-2])
      i+=1
    j+=1

  j=0
  while j<len(centers[0]):
    totald[j]=(sum(d[j]))
    vmax[j]=(max(v[j]))
    vmean[j]=(sum(v[j])/len(v[j]))
    amax[j]=(max(a[j]))
    amean[j]=(sum(a[j])/len(a[j]))
    j+=1


  # plot trajectory
  j=0
  while j<len(x):
    plt.plot(x[j],-y[j],label="airplane"+str(j))
    j+=1

  plt.title('Trajectory')
  plt.xlim([0, 450])
  plt.xlabel('x (m)')
  plt.ylabel('y (m)')
  plt.legend()
  plt.show()

  # plot velocity
  j=0
  while j<len(v):
```

```
      plt.plot(xaxis,v[j],label="airplane"+str(j))
      j+=1

    plt.ylim([0, 120])
    plt.title('Velocity')
    plt.xlabel('Time (s)')
    plt.ylabel('Velocity (m/s)')
    plt.legend()
    plt.show()

    j=0
    while j<(len(a)):
     plt.plot(xaxis,a[j],label="airplane"+str(j))
     j+=1

    plt.ylim([-50, 50])
    plt.title('Acceleration')
    plt.xlabel('Time (s)')
    plt.ylabel('Acceleration (m/s^(2))')
    plt.legend()
    plt.show()

    # plot total distance
    height = totald
    bars=[]
    i=0
    while i<(len(centers[0])):
      bars.append("airplane"+str(i))
      i+=1
    y_pos = np.arange(len(bars))
    plt.bar(y_pos, height)
    plt.title('Total distance')
    plt.xlabel('Objects')
    plt.ylabel('Distance (m)')
    plt.xticks(y_pos, bars)
    plt.show()

    # display
    j=0
    while j<len(centers[0]):
      print("Maximum velocity of object"+" "+str(j)+" "+"is"+" "+str(vmax[j])+" "+"m/s.")
      print("Mean velocity of object"+" "+str(j)+" "+"is"+" "+str(vmean[j])+" "+"m/s.")
      print("Mean acceleration of object"+" "+str(j)+" "+"is"+" "+str(amean[j])+" "+"m/s.")
      print("Maximum acceleration of object"+" "+str(j)+" "+"is"+" "+str(amax[j])+ " "+"m/s.")
      j+=1


    # visual tracking
    colors=[(255,0,0),(0,0,0),(0,255,0),(0,0,255),(255,255,0),(0,255,255),(255,0,255),(192,192

    for i in range(len(images)):
      v = Visualizer(images[i][:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=
      v = v.draw_instance_predictions(outputs[i]["instances"].to("cpu"))
      j=0
      while j<(len(centers[0])):
        cv2.circle(images[i],(coordinatestracking[i][j][0],coordinatestracking[i][j][1]),5,col
```

```
        cv2.circle(images[1],(coordinatesTracking[1][j][0],coordinatesTracking[1][j][1]),5,col
        cv2_imshow(v.get_image()[:, :, ::-1])
        j+=1
```