

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATION TECHNOLOGY

OBJECT DETECTION AND RECOGNITION SYSTEM
EMBEDDED ON A NAO ROBOT
DIPLOMA THESIS

Submitted in partial fulfillment of the requirements for the Degree of
Engineer in the domain Technology and Telecommunication Systems
Study program: Telecommunications and Information Technology

Thesis advisor(s):

Lecturer Anamaria RĂDOÎ, Ph.D
Prof. Corneliu BURILEANU, Ph.D

Student:

Diana AVRAM

Bucharest
2018

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology
Department Tc

Anexa 1

DIPLOMA THESIS
of student **AVRAM E. Diana , 441G**

1. Thesis title: Object Detection and Recognition System Embedded on a NAO Robot

2. The student's original contribution will consist of (not including the documentation part):

The diploma project consists in implementing an autonomous object detection and recognition system by a NAO robot. First, a literature review of several existing methods will be performed (for example, neural networks). The choice of the algorithms that are going to be used by the system will be done taking into account the hardware and software constraints imposed by the NAO robot (e.g., limited memory, CPU capabilities). In this sense, the student will make a comparison of the algorithms in terms of recognition performance, number of classes to be recognized, resource consumption, response time. The images containing the desired objects will be captured using either the camera that is integrated in NAO, or an external camera. The final result will be a self-contained recognition system that will signal the existence of a certain object (e.g., recorded voice / movement, terminal output). In addition, the algorithms will be customized to the user needs.

3. Pre-existent materials and resources used for the project's development:

Programming Language: Python, C++; Hardware: NAO Robot, possibly an external camera; Software libraries: OpenCV; Simulation program: Coregraphe

4. The project is based on knowledge mainly from the following 3-4 courses:

DEPI, uC, Digital Signal Processing, OOP

5. The Intellectual Property upon the project belongs to: U.P.B.

6. Thesis registration date: 2017-11-28 15:18:23

Thesis advisor(s),

Ș. L. Dr. Ing. Anamaria RĂDOI

signature:

Prof. Dr. Ing. Corneliu BURILEANU

signature:

Departament director,

Conf. dr. ing. Eduard POPOVICI

signature:

Validation code: **c4c3318b5c**

Student,

signature:

Dean,

Prof. dr. ing. Cristian NEGRESCU

signature:

Statement of Academic Honesty

I hereby declare that the thesis "*Object Detection and Recognition System Embedded on a Nao Robot*", submitted to the Faculty of Electronics, Telecommunications and Information Technology in partial fulfillment of the requirements for the degree of Engineer in the domain *Technology and Telecommunications Systems*, study program *Telecommunications and Information Technology* is written by myself and was never before submitted to any other faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. Text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from the respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, July 2018

Diana AVRAM

A handwritten signature in blue ink, appearing to read 'Avram', is written over a horizontal line.

Table of Contents

Table of Contents.....	7
List of Figures.....	9
List of Tables	11
List of Abbreviations	13
Acknowledgments	15
CHAPTER 1 Introduction	17
1.1 Motivation.....	17
1.2 The robot Nao.....	17
1.3 The proposed solution.....	21
CHAPTER 2 Nao's Recognition Module.....	23
2.1. NAOqi Framework.....	23
2.2. Choregraphe.....	25
2.3 Recognition workflow.....	26
2.4. Limitations.....	28
CHAPTER 3 Recognition using convolutional neural networks.....	29
3.1. Neural network.....	29
3.1.1 Perceptrons.....	29
3.1.2 Architecture of a Feed-Forward Neural Network.....	30
3.1.3 Multi-layer Perceptron.....	31
3.1.4 Activation functions.....	32
3.1.5 Back-propagation Algorithm.....	33
3.2 Convolutional Neural Networks.....	40
3.3 Mobile Nets.....	48
3.3.1 Introduction.....	49
3.3.2 Architecture.....	49
3.3.3 Advantages.....	51
3.4 Single Shot MultiBox Detector.....	52
3.4.1 Introduction.....	52
3.4.2 Architecture.....	54
3.4.3 MultiBox.....	55
3.4.4 Intersection over Union ratio.....	55
CHAPTER 4 The Implemented Vision Algorithm.....	57
4.1 Working principle.....	57
4.2 Object detection using MobileNet SSD	62
4.2.1 Real time object recognition model, using Tensorflow.....	62
4.2.2 Object recognition model, using Caffe.....	67

CHAPTER 5 Experimental Results.....	73
5.1. Results on Nao.....	73
5.2. The performance of the Convolutional Network.....	78
CHAPTER 6 Conclusions and Future Steps.....	81
6.1 General Conclusions.....	81
6.2 Personal contributions.....	82
6.3 Future work.....	82
References.....	83

List of Figures

Figure 1.2.1 - Technical features of Nao	18
Figure 1.2.2 - Mobility & equilibrium of Nao.....	19
Figure 1.2.3 - Nao's actuators.....	20
Figure 2.1.1 - The relation between brokers, modules and the corresponding methods.....	24
Figure 2.2.1 - Parallel actions by Nao in Choregraphe.....	25
Figure 2.2.2 - Series actions by Nao in Choregraphe.....	25
Figure 2.3.1 – ALVisionRecognition Module Workflow.....	26
Figure 2.3.2 – Parameters of PictureDetected.....	27
Figure 3.1.1.1 – Working principle of the perceptron.....	30
Figure 3.1.2.1 – An example of a feed-forward neural network, with one hidden layer.....	31
Figure 3.1.4.1 – Neural network with one hidden layer.....	32
Figure 3.1.4.2 – Graphical representation of the activation functions.....	33
Figure 3.1.5.1 a – Gradient descent for cost function $\mathcal{C}(\mathbf{w})$	35
Figure 3.1.5.1 b – Gradient descent for cost function $\mathcal{C}(\mathbf{w}, \mathbf{b})$	35
Figure 3.1.5.2 a - Big learning rate.....	37
Figure 3.1.5.2 b - Small learning rate.....	37
Figure 3.1.5.3 – Sample of MNIST dataset.....	37
Figure 3.1.5.4 - Neural network with one hidden layer.....	38
Figure 3.1.5.5 - Small parts of digit “4”.....	39
Figure 3.2.1 – Regular Neural Network vs. Convolutional Neural Network.....	40
Figure 3.2.1 – An example of ConvNet.....	41
Figure 3.2.2 - The representation of a picture by a matrix of numbers.....	42
Figure 3.2.3 - The greyscale image to be analyzed and its corresponding matrix, G.....	42
Figure 3.2.4 - The convolution process.....	43
Figure 3.2.5 - Convolution with three filters.....	44
Figure 3.2.6 - Example of MaxPooling.....	45

Figure 3.2.7 - The fully connected layers from the network given as example in Figure 3.2.1	45
Figure 3.2.8 - Visualization of a ConvNet's layers.....	46
Figure 3.3.2.1 – Splitting standard convolution into depthwise and pointwise convolutions.....	49
Figure 3.4.1 - Detection examples of SSD on personal dataset.....	53
Figure 3.4.4.1 Dog detection example - the corresponding ground-truth and predicted bounding boxes.....	55
Figure 4.1.1 – Detection algorithm on Nao, with pictures taken with a 12 MP camera (on smartphone).....	58
Figure 4.1.2 – Detection algorithm on Nao, with pictures taken with an 8 MP camera (on smartphone).....	59
Figure 4.1.3 - The recognized classes by Nao.....	60
Figure 4.1.4 - General flow diagram of the program.....	61
Figure 4.2.1.1 (a) - COCO 2017 object categories, labels not shown.....	62
Figure 4.2.1.1 (b) - COCO dataset examples.....	62
Figure 4.2.1.2 – Results from real time detection.....	64
Figure 4.2.1.3 – Unsuccessful results from real time detection.....	65
Figure 4.2.2.1 - Successful and unsuccessful recognitions (computer implementations)	68
Figure 5.1.1 – Successful recognitions on Nao.....	73
Figure 5.1.2 - Successful real time recognitions, on Nao.....	76
Figure 5.2.1 - Example of test pictures.....	78
Figure 5.2.2 – Successful vs. unsuccessful <i>people</i> recognition.....	80

List of Tables

Table 1.2.1 - Degrees of freedom for Nao.....	20
Table 2.4.1 – Limitations of ALVisionRecognition module.....	27
Table 3.1.4.1 – Activation functions.....	33
Table 3.2.1 - Layers’ dimensions for a CIFAR-10 input image.....	41
Table 3.2.2 - Comparison between different architectures, considering the dataset used for training, the number of parameters and the validation error.....	47
Table 4.1.1 - Parameters of <i>ALPhotoCapture</i> module.....	58
Table 4.2.1.1 - Precision and Accuracy for the models provided by Google’s API.....	63
Table 5.2.1 - Results from the Caffe model.....	78

List of Abbreviations

API - Application Programming Interface

CAFFE - Convolutional Architecture for Fast Feature Embedding

CIFAR - Canadian Institute For Advanced Research

CNN - Convolutional Neural Network

COCO - Common Objects in Context

ConvNet - - Convolutional Neural Network

CPU - Central Processing Unit

FPS - Frames per Second

GPU - Graphics processing unit

ILSVRC - ImageNet Large Scale Visual Recognition Challenge

IP - Internet Protocol

MNIST - Modified National Institute of Standards and Technology

OpenCV - Open Source Computer Vision Library

SSD – Single Shot Detection

VGA - Quarter Video Graphics Array

VOC - Visual Object Classes

Acknowledgements

I want to thank my coordinators, Lecturer Anamaria RĂDOI Ph.D and Prof. Corneliu BURILEANU, Ph.D for their support in the practical and the theoretical part of this thesis. I am very thankful for their moral support and also for the equipment they provided me in the laboratory, for achieving my goal in this project.

In addition, I am deeply grateful to Eng. Georgian NICOLAE and Eng. Ana-Antonia NEACȘU for their introduction in the world of Artificial Intelligence, as well as for Georgian's guidance throughout the practical implementation.

1

INTRODUCTION

1.1 Motivation

With the high evolution nowadays in the robotics and autonomous systems area, the robots need to be skillful in a particular domain. Achieving this means getting information about the environment through sensors; and since vision is the most important sense, offering up to 80% of all impressions by means of our sight for humans, then one of the richest and most useful information is the one captured by cameras.

Developing algorithms for detecting objects with high accuracy remains a challenge and a domain of research in robotics. And this is a problem of utmost importance, since only by having a good perception of the surroundings, will the robots be able to perform specific tasks.

1.2 The robot Nao

General aspects

Nao is the most used humanoid robot for education worldwide; its friendly aspect, mobility and possibility to be fully programmed makes Nao a powerful tool in the education and research areas like helping children develop their robotics appetite, aid for autistic children in therapy modules and laboratory assistants.

The humanoid robot is designed to be personalized, depending on the wanted application. Not only does it have several sensors and motors that can be manipulated by the user, but it also has a user-friendly programming software, which enables an easy interaction with these sensors and motors. Thus, using Choregraphe, Nao can be programmed to have a particular personality.

The robot was firstly presented in 2006 and since then, it continued to develop. It was firstly designed with the idea of helping children with mental disabilities in mind and this project aims to make a step forward in this direction. With a view to helping Nao become more autonomous and

implicitly making the interaction with the robot more enjoyable, an object detection and recognition system embedded on the robot was developed.

Technical details

While working on the main program, I encountered several limitations, imposed by the robot; and since the application was intended to be embedded from the beginning, the restrictions needed to be acknowledged. Some technical features are presented in Figure 1.2.1.

Currently in the fifth version, about 10 000 Nao robots have been sold around the world, in educational institutes from over 70 countries. The new generation of Nao has a 1.6 GHz processor, two HD cameras and a height of 58 cm; also, its big advantage is its humanoid look, as it can be observed from Figure 1.2.1. The robot has 4 microphones, through which it can communicate in 8 languages; apart from these, it is fitted with a distance sensor, two infrared emitters and receivers, 9 tactile sensors and 8 pressure sensors.



Figure 2.2.1 - Technical features of Nao

The robot's shape, along with its capacity to move, thanks to its 25 degrees of freedom allow it to move, to maintain its equilibrium and to be aware of its standing or lying position; this equilibrium ability is presented below.



Figure 1.2.2 - Mobility & equilibrium of Nao

Available resources

With 58 cm height and 28 cm width, the 4.5 kg robot is able to perform complex moves, from moving forward and backwards, sitting down and getting up, to stretching and maintaining equilibrium on a single foot, as shown in the Figure 1.2.2.

Battery

Nao has a Lithium Ion battery with nominal voltage 21.6 V, autonomy of 90 minutes at normal use and a charging duration of 3 hours; the robot can be used when it is plugged in.

Motherboard

The fourth version of Nao - which I used in this project – has two main processors, one at the head level and one at the torso level.

- ❖ The one at the head level is an Intel x68 processor, ATOM Z530, with a cache memory of 512 KB; the clock rate is 1.6GHz for the 32 bits instruction set. It is a single core processor, designed mainly for mobile applications and it has the advantage its power efficiency. Additionally, ATOM Z530's architecture is based on Bonnell's microarchitecture, thus being able to execute two instructions per cycle.
- ❖ The processor from the torso level is an ARM7TDMI type, which controls the actuators (that move the robot). The ARM7TDMI microcontroller has a 32 bits instruction set, of type RISC, also offering good performances for reduced power consumption. The local microcontrollers from the actuators are Microchip 16-bit dsPICS and they communicate with the second CPU by two buses, RS-485 type, at a throughput of 460 Kb/s.

Audio

The robot has a stereo system, comprised of two speakers on each side of the head and 4 microphones also placed on the head, having a frequency range of 300Hz – 8KHz.

Video

On the front side of Nao, two identical video cameras are placed, which offer images of 1280x960 resolution, at 90 frames per second. They are used to identify objects around and the space where Nao moves.

Actuators

The actuators are the motors that permit the robot to move; they are placed at joints and Nao has 25 actuators, hence its great mobility.

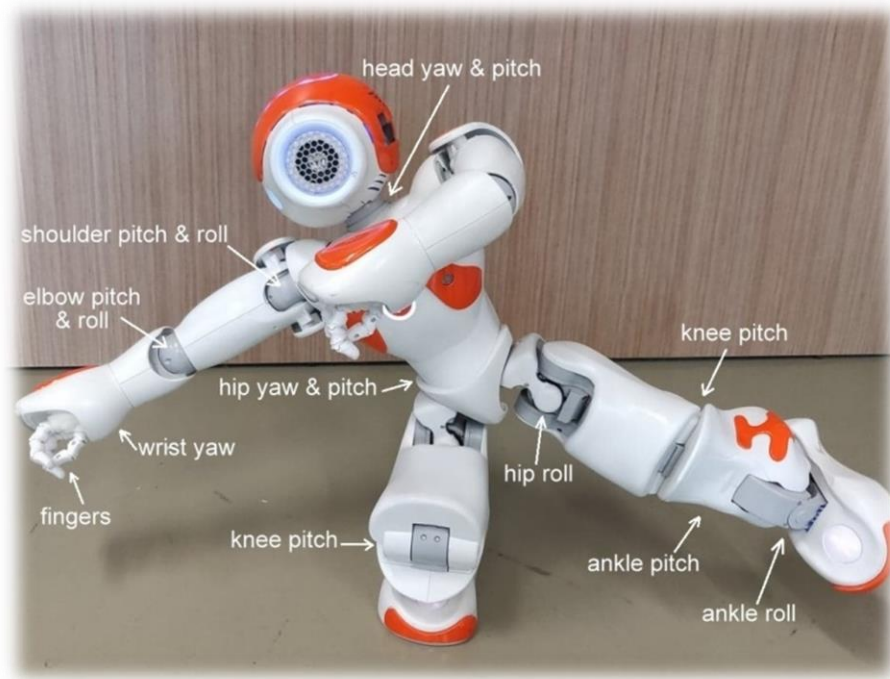


Figure 1.2.3 - Nao's actuators

The multitude of the robot's actuators placement is presented in Figure 1.2.3; these offer Nao 25 degrees of freedom. So, there are 25 independent parameters that define Nao's state, starting from its head to its feet, as follows:

Location	Degrees of freedom	Observations
Head	2	the head can yaw and pitch
Arm	10	5 in each arm: 2 at shoulder, 2 at elbow, 1 at wrist level
Hand	2	1 in each hand, for hand grasping
Pelvis	1	-
Leg	10	5 in each leg: 2 at ankle, 1 at knee, 2 at hip level

Table 1.2.2 - Degrees of freedom for Nao

1.3 The proposed solution

There are many projects considering the object recognition problem and each is particularized for its specific task, considering different conditions and training datasets. This project aims to propose a way to detect and recognize objects for the Nao robot. This implies the robot taking several pictures using its front camera and applying an algorithm, in the end outputting through its microphone the identified objects. This local processing has the great advantage of independence from other resources, like an external camera or memory, but there are also some limitations that need to be considered.

Particularly, the processed images have a limited resolution, of $640\text{ px} \cdot 480\text{ px}$ and the processing resources are also limited, namely the processor and the memory. The user has limited access to resources, specifically an ATOM Z530 1.6 GHz CPU and a 1 GB of RAM. In addition, a very big disadvantage of working on the robot is that it does not support some libraries used in every computer vision application, such as OpenCV3, or frameworks like TensorFlow.

Consequently, a compromise needs to be made between the computational power of the robot, its limited memory and a good network architecture, that produces reasonable results.

2

NAO'S RECOGNITION MODULE

2.1. NAOqi Framework

In order to make the work with the robot easier and more user-friendly, the NAOqi framework was created. Its role is to offer the possibility of working with multiple modules that the robot has, like video, audio and motion modules, keeping in mind the need for parallelism, synchronization and events, used for a robot. In this way, the user can access the needed resources, while sharing information between them.

This framework allows *introspection*; this means that the framework knows the available functions from the modules and where to look for them. So, the robot knows what are the available API functions.

NAOqi Modules APIs

- ❖ The robot has several modules APIs:
- ❖ Motion – offers methods which allow Nao to move, by controlling the joint stiffness and position, or to walk
- ❖ Audio – offers modules to play or record audio files, detect sound events and even speech recognition and text to speech methods
- ❖ Sensors – deals with the bumpers, tactile hands and head, the battery and LEDs
- ❖ Vision - offers modules for detection of: backlighting , darkness, faces, movement, photos and red balls.

In order to access the NAOqi Modules, a broker is needed; its scope is to load libraries containing the wanted modules and also to provide directory services and network access for calling them. The hierarchy is presented below, in Figure 2.1.1.

To work with modules, *proxies* are needed, that act like the corresponding modules. So, to make the robot able to react to defined circumstances, modules can subscribe to events. For instance, if the robot listens for the user's words, in order to recognize some specific ones from a vocabulary, a proxy needs to be subscribed to "WordRecognized". After this, the reaction is specified in case of successful recognition.

Broker

This is an object that has two roles: it provides directory services (allowing the user to find *modules* and *methods* – see Figure 2.1.1) and works transparently.

Proxy

This is an object that behaves as the module it represents. Concretely, if the user creates a proxy to the ALMotion module, they get an object that contains all the ALMotion methods. The creation of a proxy is simple: the user has to use the name of the module in their code (local call). More details about this can be found in the NAO Software 1.14.5 documentation about NAOqi Framework [1].

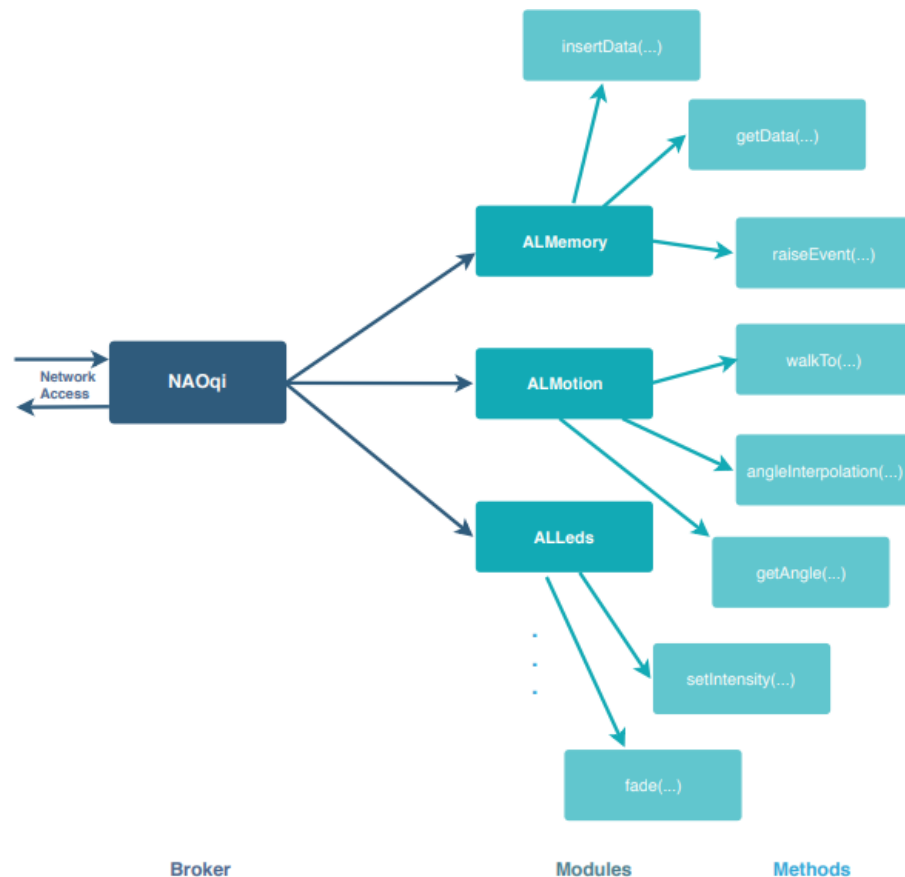


Figure 2.1.1 - The relation between brokers, modules and the corresponding methods

From the vision class, apart from the mentioned modules, the robot has one which is noteworthy, because it offers vision learning and recognition capabilities: ALVisionRecognition

ALVisionRecognition

The scope of this module is to make Nao recognize previously learned objects or pictures.

Working principle

This vision module is easy to work with using Choregraphe.

2.2. Choregraphe

This is a user-friendly multi-platform desktop application, developed by Aldebaran Robotics, that allows programming of the robot. This graphical environment enables the user to make connections of high level behaviors easily, by using specific behavior boxes. Apart from this, it gives the possibility for fine tuning of joint motions. Finally, at the lowest level, this application allows programming in Python.

Choregraphe provides the user with the NAOqi functions, in a friendly way. Thus, the user can execute some predefined behaviors, by linking some specific boxes. For instance, if the user wants the robot to execute some chained actions, they have to link the according boxes sequentially; if they want to execute several behaviors in the same time, the boxes need to be linked parallel. An example of a series and parallel linking, to obtain a more complex behavior is presented in Figures 2.2.1 and 2.2.2.

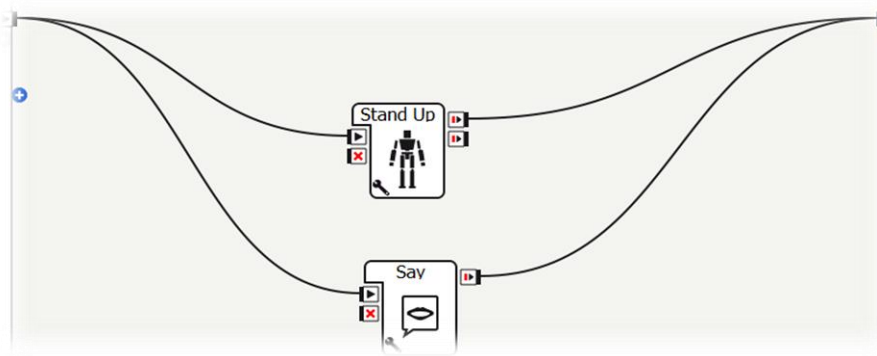


Figure 2.2.1 - Parallel actions by Nao in Choregraphe

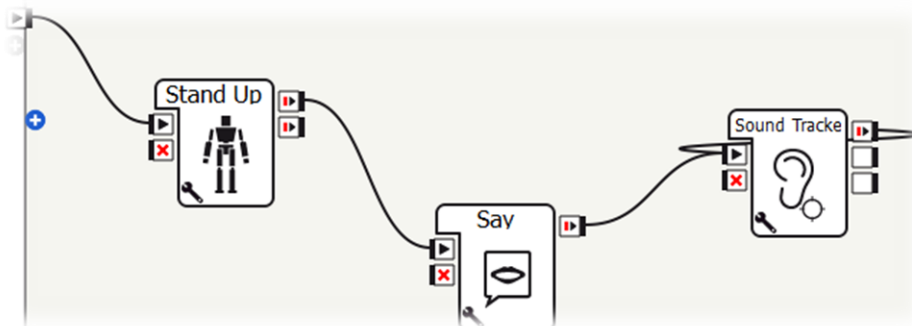


Figure 2.2.2 - Series actions by Nao in Choregraphe

Figure 2.2.1 presents parallel actions performed by the robot in Choregraphe, specifically standing up and saying “*Hello, I will present you my multitasking ability through Choregraphe*”; and Figure 2.2.2 shows series actions performed in a similar manner, like standing up, then saying “*Now, I am going to track the sound in this room*” and next performing sound tracking until stopped.

To conclude, working in Choregraphe can be intuitive. One of the interesting modules Nao has is ALVisionRecognition and some tests were made, in order to analyze it.

2.3 Recognition workflow

ALVisionRecognition module aims to recognize objects, depending on previously learned models. Below is presented a diagram of how this module briefly works:

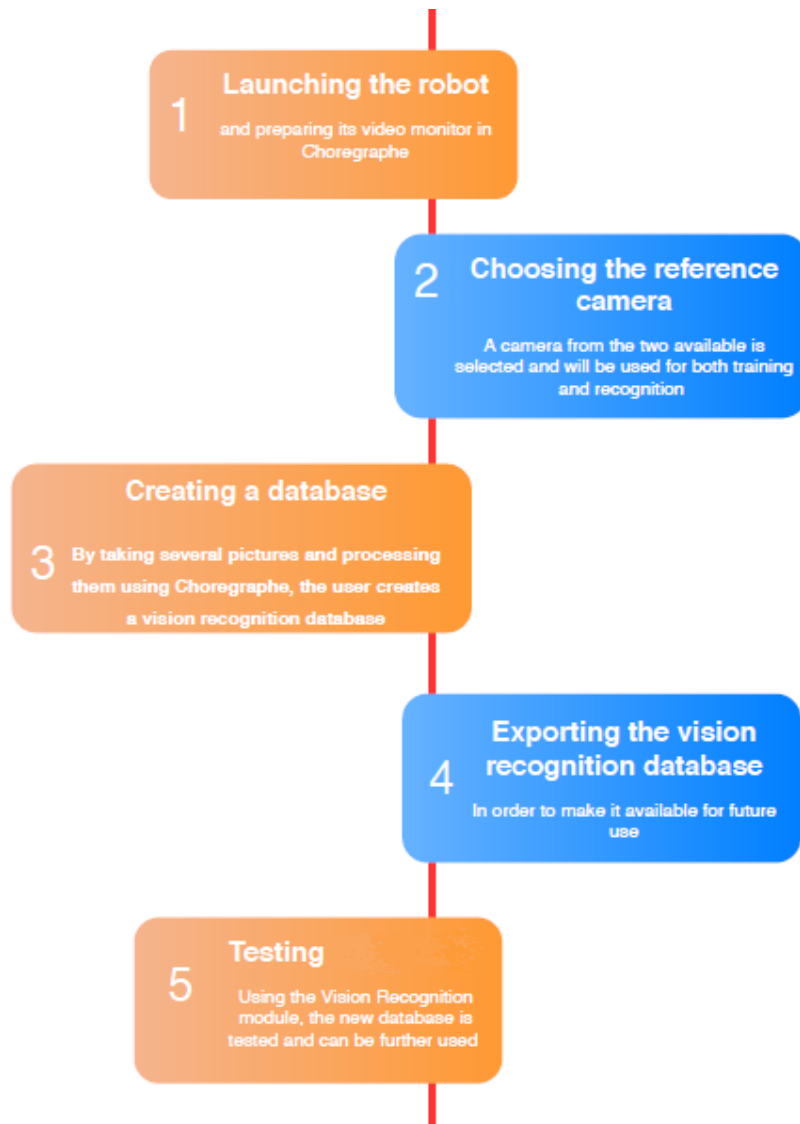


Figure 2.3.1 – ALVisionRecognition Module Workflow

Teaching the robot to recognize specific objects

The robot learns images, objects and pictures using its video monitor. For this, the user has to pursue the following steps:

- ❖ connect to a robot – either real or virtual
- ❖ access the *video monitor* - in order to see what the robot's camera sees
- ❖ execute the *learn* command - now, the user is given 4 seconds to place the object in a desired position relative to the robot's camera; after this, a capture is taken, by switching to QVGA resolution

- ❖ execute the *draw* command - in this moment, the user is able to draw the contour of the object, segment by segment, in order to differentiate the object from its background
- ❖ label the new object and select the corresponding side
- ❖ export the vision recognition database on the robot

After the vision recognition database is imported on Nao, a vision recognition box needs to be created using Choregraphe and tested. If the user does not want to use this environment, they can also access the `ALVisionRecognition` module from Python. The steps are described in the Aldebaran documentation [1].

Similar with other extractor modules, the results from the object recognition are placed in the `ALMemory`. The webpage of the robot can be accessed in a browser, using its IP and by selecting `Advanced -> Memory -> PictureDetected`; when something is recognized, the `ALValue` changes and its parameters are explained in the following diagram:

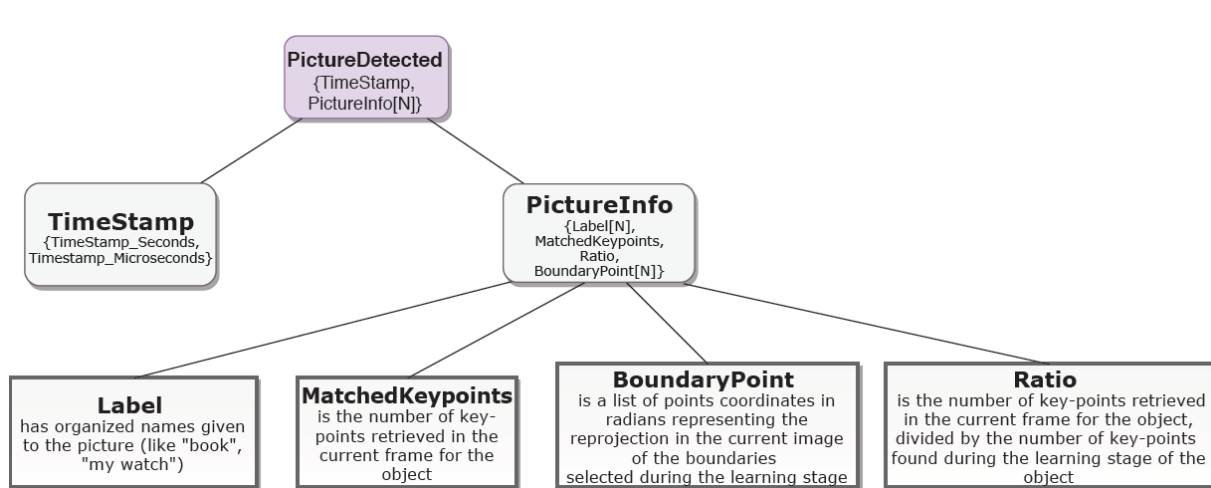


Figure 2.3.2 – Parameters of PictureDetected

2.4. Limitations

Although the idea of making the robot recognize previously learned object is a good one, the module has big limitations. Firstly, it enables the recognition of key points in a capture taken by the robot, thus making the real-time recognition very difficult. Also, the module is only useful for recognizing objects that were previously learned; therefore, the robot cannot perform in this area when shown a different entity from the same class. Last, but not least, a great disadvantage is that for a successful recognition, the object needs to be in the same conditions as in the training process. The limitations are presented in the table 2.4.1.

Limitation	Additional information
distance	it must to be between half and twice the distance used for learning
light conditions	-
angles	the inclination needs to be less than 50°
rotation	-

Table 2.4.1 – Limitations of ALVisionRecognition module

Apart from these, the module has other restrictions:

- ❖ It cannot recognize untextured pictures – because the recognition is based on key points, not on the object's shape. Also, after training, the database is exported on the robot and for better performance; this contains only essential information for detection.
- ❖ At the moment, as previously mentioned, it is not able to recognize object classes (for instance a person), but instances (in this example, a person in general).
- ❖ Several learning processes reduce the detection rate – this happens, because the detection algorithm on the robot works in the following way: every detected key point in the current image is compared with one point from the database; and if two scores for choosing between two classes are too close, the key point is dropped and it is not associated with any of the two possible objects.

3

RECOGNITION USING CONVOLUTIONAL NETWORKS

3.1. Neural network

An artificial neural network is a computational brain-inspired model, intended to replicate the way humans learn. Since they are similar to the way biological neural networks in the human brain process information, the neural networks comprise of input, output and hidden layers; the latter ones are meant to transform the input (for instance, an image) into something that the output can use (like a label). This is why they are powerful tools for finding patterns that are too complex for the human perceptions.

3.1.1 Perceptrons

The perceptron is an algorithm of supervised learning of binary classifiers, firstly developed in 1957 by Frank Rosenblatt, in the Cornell Aeronautical Laboratory. The perceptron was intended to be a machine (not an algorithm) in the beginning and it was used for image recognition. After a period of stagnation, the idea bloomed again, when it was admitted that a multilayer perceptron would be more efficient.

From another perspective, a perceptron is a single layer neural network and a multi-layer perceptron is called a neural network.

The working principle is the following: a perceptron takes the input – which is represented by more binary inputs -, processes it and produces one single output, as in Figure 3.1.1.1.

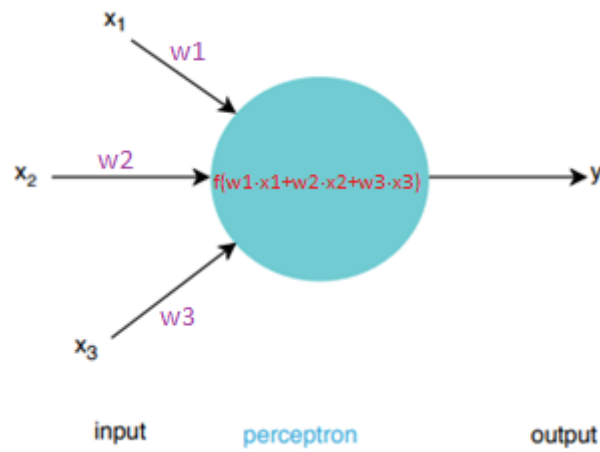


Figure 3.1.1.1 – Working principle of the perceptron

In this example, the perceptron takes three inputs, x_1 , x_2 and x_3 and outputs y ; of course, the number of inputs can be larger. In order to compute the output, Rosenblatt had the following idea: he proposed giving each input some *weights* (w_1 , w_2 and w_3 in the analyzed case); these are real numbers, which express the degree of importance of the input, with respect to output. Following, the binary output is determined depending on the sum $\sum_{j=1}^n w_j \cdot x_j$: if it is less or equal than a *threshold*, the output takes one value and if it is greater than the proposed threshold, the output takes the other value. As a remark, the threshold is a real number as well, which characterizes the neuron. Concretely, this idea can be written as:

$$\begin{aligned} &\text{if } \sum_{j=1}^n w_j \cdot x_j < \text{threshold} \\ &\quad \text{out} = 0; \\ &\quad \text{else} \\ &\quad \text{out} = 1; \end{aligned} \tag{1}$$

To make a parallel with the human brain, the perceptron can be regarded as a model of decision making. Using this idea, an efficient way of making a decision is by considering all the factors as inputs and assign each of them a level of importance, that is a weight. Depending on the chosen threshold, an optimal decision – the final output – can be taken, considering all variables.

Of course, taking a final decision based only on one operation is the simplest case; a complex network of perceptrons can take more subtle decisions.

3.1.2 Architecture of a Feed-Forward Neural Network

A *feed-forward neural network* is the simplest artificial network; it has more *neurons* (also called nodes) arranged in *layers*. While the nodes from adjacent layers have *connections*, each connection has an associated weight.

An example of such a network is presented in Fig. 3.1.2.1.

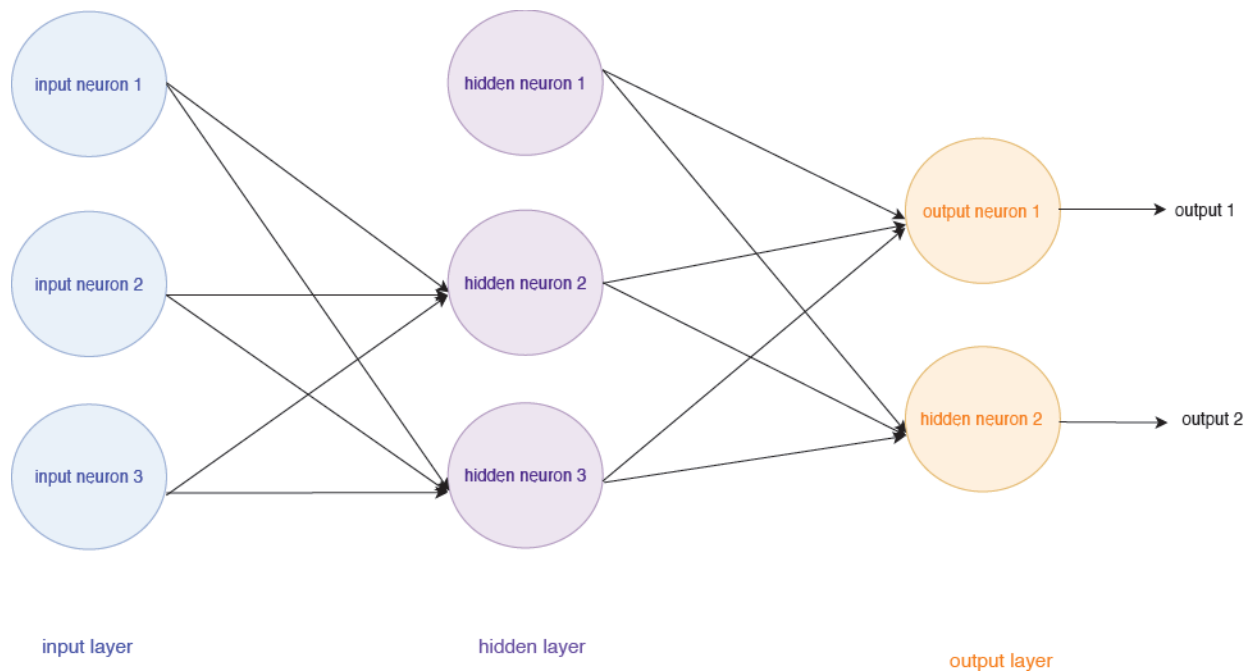


Figure 3.1.2.1 – An example of a feed-forward neural network, with one hidden layer

Thus, a feed-forward neural network consists of the following types of layers:

- ❖ Input layer – the input nodes provide the following layers information about the outside world; no computation is performed at this layer; also, the input nodes are mandatory in a neural network.
- ❖ Hidden layer – the nodes from this layer do not have a direct connection with the outside world; they perform computations and transfer information from input to output; unlike the input or output nodes, the hidden ones may or may not be present in a feed-forward network.
- ❖ Output layer – it is responsible for computations and transfer of information from the network to the outside world

Obviously, in a feed-forward neural network, the information travels in only one direction: from input to output; so there are no loops or cycles. Examples of this type of network are the single layer perceptron, presented in the previous paragraph of this section and also the multi-layer perceptron, which shall be discussed in the following section.

3.1.3 Multi-layer Perceptron

This type of network contains one or more hidden layers and unlike the single layer perceptron, this one can learn non-linear function.

An example of such a network is shown in Figure 3.1.1.2. In that network, the first layer – which is the first column of perceptrons – makes very simple decisions, by analyzing the input. Afterwards, the second layer takes as input the output of the first layer and takes a decision, at a more complex level than the first one. Following this rule, multi-layer networks can be imagined for more sophisticated problems. Of course, the output layer provides the final result.

Mathematically, the multi-layer network problem can be written as:

$$\text{if } \sum_{j=1}^n w_j \cdot x_j + \text{bias} < 0$$

$$\begin{aligned} & \text{out} = 0; \\ \text{else} & \\ & \text{out} = 1; \end{aligned} \tag{2}$$

The bias = - threshold and it measures how easy it is to get the perceptron to output a one. So, the bigger the bias, the easier it is for the perceptron to output a one; and of course, for a negative bias, it is difficult for the perceptron to output 1.

3.1.4 Activation functions

The process of creating an efficient learning algorithm is difficult and this is why making some adjustments during this process is a very good idea. Specifically, let us take the example of having some images of animals provided and an animal classification network is needed. In order to see how the process of learning works, some small changes in the weights or bias are made; it is desired that one small change produces a small corresponding change in the network's output. Schematically, this is what is wanted:

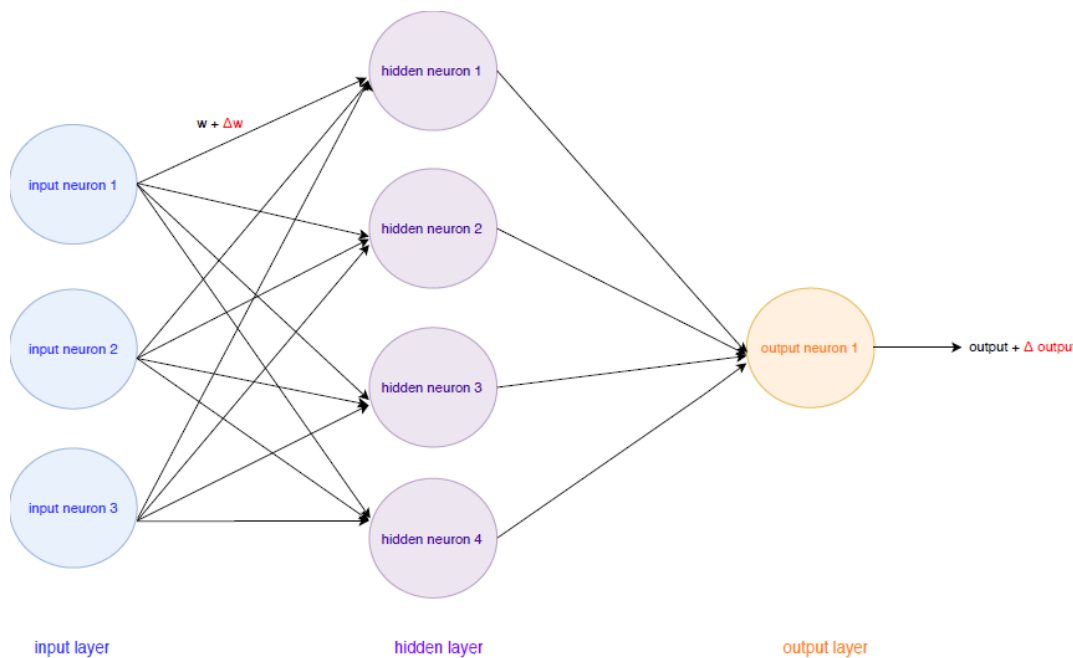


Figure 3.1.4.1 – Neural network with one hidden layer

As Figure 3.1.4.1 suggests, a small change in any weight or bias causes a small change in the output.

But sometimes, a small change in the weights or biases can cause the output to wrongly flip from one value to the other; and that flip can cause the next layer to change its parameters, thus changing the behavior of the whole network.

To solve this problem, a neuron is introduced, that performs a certain fixed mathematical function operation: sigmoid, tanh or ReLU. They have the following characteristics:

Activation functions		
Name	Operation	Range
Sigmoid	$\sigma(x) = 1 / (1 + \exp(-x))$	[0, 1]
Tanh	$\tanh(x) = 2\sigma(2x) - 1$	[-1, 1]
ReLU	$f(x) = \max(0, x)$	[0, ∞]

Table 3.1.4.1 – Activation functions

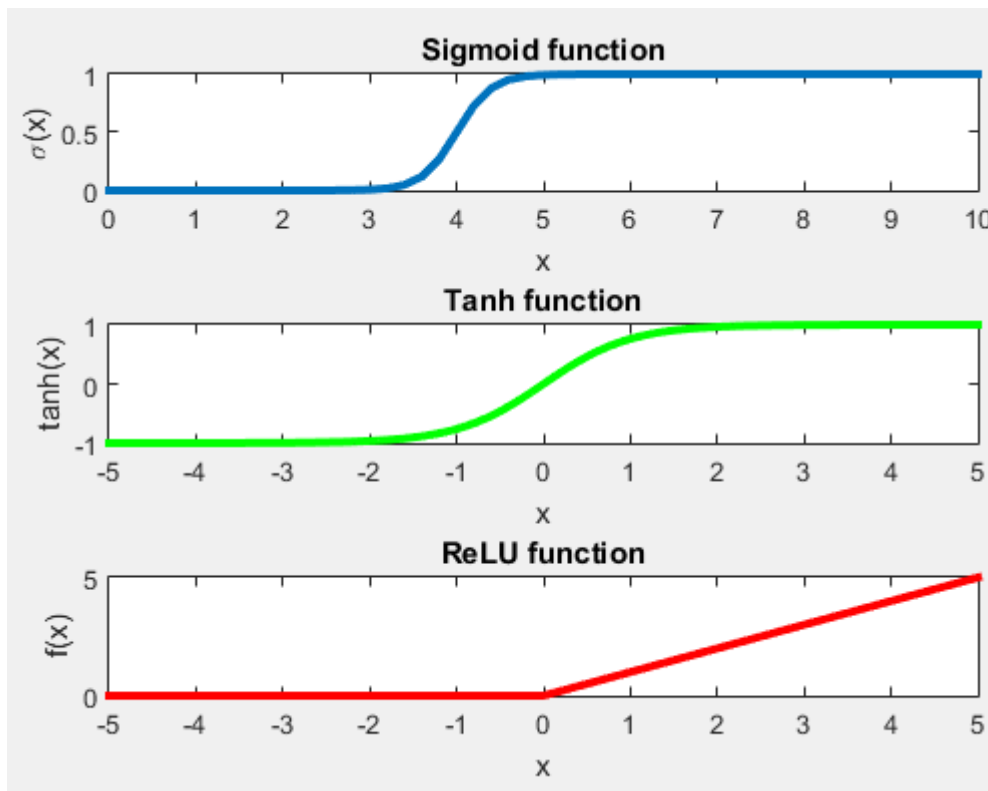


Figure 3.1.4.2 – Graphical representation of the activation functions

These functions take as input real values and shrink the range to a certain interval, specified Table 3.1.4.1.

3.1.5 Back-propagation Algorithm

This is the process by which an artificial Neural Network is trained. As it will be discussed in the following sections, neural networks can learn their *weights* and *biases*, using an algorithm, *gradient descent*, which computes the *gradient* of the *cost function*; and the algorithm to do this is known as *back-propagation*.

The base of this algorithm is an expression for the partial derivatives of the cost function C , with respect to any weight or bias in the network, $\frac{\delta C}{\delta w}$. Mathematically, this expression gives information about how fast the cost changes, when the weights change; practically, back-propagation gives

details about how the overall behavior of the network is affected by the weights and biases changing.

Cost function

Let us consider a neural network and we desire an algorithm, which finds weights and biases, in such a way that the output from the network approximates $y(x)$ (the correct output), for all training inputs, x , as good as possible. In order to quantify how well this goal is achieved, a *cost function* is defined. This is also referred as loss function and intuitively, it can be defined as the following:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a(x, w, b)||^2 \quad (3)$$

where:

- ❖ w - represents all the weights in the network;
- ❖ b - represents all the biases;
- ❖ n - is the total number of training inputs;
- ❖ a - is the vector of outputs from the network, when x is input and the sum is over all inputs;

So, as the notation suggests, the cost function will be given by the length of the vector resulting from the summation and squaring it. For this, C is called *mean square error*. The reasons to choose this specific cost function will be presented in the following paragraph.

First of all, the mean square error function is always positive, since every term is squared. Therefore, its minimum is easy to be found. In addition, the cost decreases as the output of the network, a , becomes closer to the correct output, $y(x)$; in this way, this function can find weights and biases so that $C(w, b) \approx 0$.

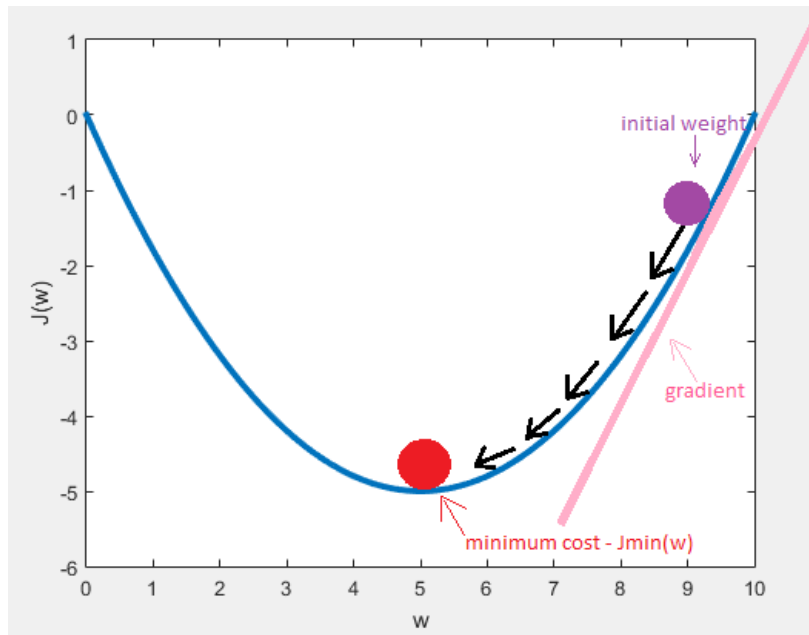
This is a very good result, because knowing this, the algorithm can be thought to minimize the cost function; and since that depends on the weights and biases, these two can be finally found in an optimized form.

As a remark, the choice of the cost function depends on the application. It can be adjusted and different minimizing weights and biases can be obtained, but in this paper, the cost function from equation 3 shall be presented.

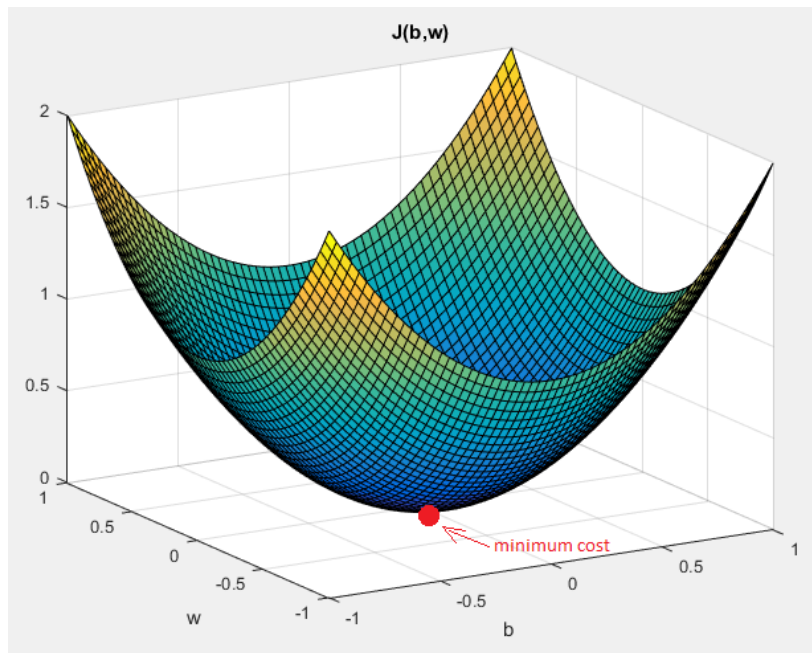
To conclude, the cost function is used to monitor the predictions' error. This is why minimizing it means getting the lowest value of the error, which means increasing the accuracy. The algorithm that does the minimization is the *gradient descent*.

Gradient descent

This is an algorithm that minimizes the cost function, by getting the lowest error value. Graphically, if C was dependent only on one variable, the algorithm would be similar to the one in Fig. 3.1.5.1 a.



(a)



(b)

Figure 3.1.5.1 (a) – Gradient descent for cost function $C(w)$. (b) Gradient descent for cost function $C(w, b)$

If the cost function depends on two variables: the weight and the bias, the representation can be similar to Figure 3.1.5.1 b. Furthermore, for cases with dependency on more variables, the principle applies similarly, but the representation and visualization of the results is more difficult. So, for a better understanding, the intuitive case with the cost function dependent on two variables shall be analyzed further.

For the simpler cases, the gradient can be understood as the slope of a function, as suggested in Figure 3.1.5.1 a. If the gradient is high, the slope will be steep and the model will be learned faster. But this presents a risk, which will be discussed in the next subchapter, Importance of learning rate.

To generalize, the gradient is the partial derivative of the cost function, with respect to all inputs.

From the graphical representation of the function in Figure 3.1.5.1 b, the minimum of the function is obvious and easy to spot, but the function for more complicated problems will have a more complex graph. But for that case, the gradient descent rule can be explained; if a random point is chosen, the problem can be seen as a ball rolling down a valley, until the minimum point where it stabilizes. This simulation is made by computing the partial derivatives of C , because they give information about the local shape of the valley and what trajectory should the ball have. Concretely, if the ball moves by a small amount Δv_1 in the v_1 direction and Δv_2 in the v_2 direction, then the cost function will be:

$$\Delta C = \frac{\delta C}{\delta v_1} \cdot v_1 + \frac{\delta C}{\delta v_2} \cdot v_2 \quad (4)$$

In this equation, w and b were replaced by $v = [v_1, v_2]$

For simplification in writing, the following notations are made:

❖ the vector of changes:

$$\Delta v = (\Delta v_1, \Delta v_2)^T \text{ (transposed matrix)} \quad (5)$$

❖ the gradient vector:

$$\nabla C = \left(\frac{\delta C}{\delta v_1}, \frac{\delta C}{\delta v_2} \right)^T \quad (6)$$

Considering these notations, equation (4) can be rewritten as:

$$\Delta C = \nabla C \cdot \Delta v \quad (7)$$

Thus, from equation (7), it is intuitive how to choose Δv , in order to make ΔC negative. In particular, if we denote η as the learning parameter, it can be chosen as:

$$\Delta v = -\eta \cdot \nabla C \quad (8)$$

In this way, equation (7) becomes:

$$\Delta C = \nabla C \cdot \Delta v = \nabla C \cdot (-\eta \cdot \nabla C) = -\eta \cdot \|\nabla C\|^2 \leq 0; \quad (9)$$

So, the cost function will always decrease, if v changes according to (8).

In the parallel with the ball on the hill, the ball's move is described by the equation:

$$v_{final} = v_{initial} - \eta \cdot \nabla C, \quad (10)$$

Which is the adaptation of the original formula in (8).

To sum up, the gradient descent repeatedly decreases C , until a global minimum is reached.

The same formulas apply if C is a function of several variables, n .

Importance of learning rate

This parameter represents how big are the steps that the gradient descent takes, in order to find the minimum. So, this parameter determines how slow or fast the function will reach the optimal weights. Two limit cases are distinguished; either the step is too big, or too small.

- ❖ if the learning rate is too big, it is possible that the gradient descent algorithm does not reach the local minimum, but only oscillates around it, like in Figure 3.1.5.6 a;
- ❖ if η is too small, the changes in Δv will also be small, therefore the minimum of the cost function will be found after a long time; Figure 3.1.5.6 b illustrates this situation.

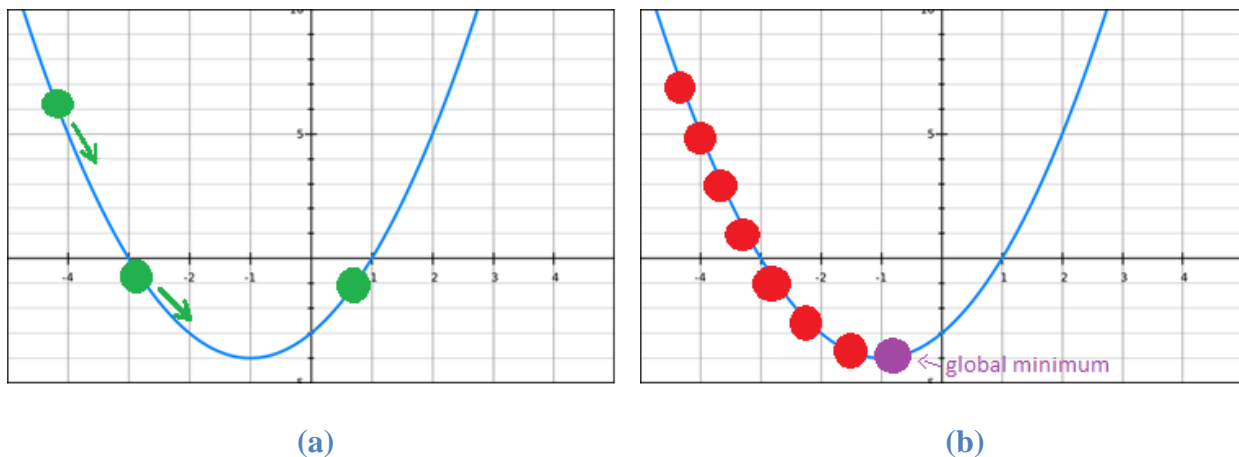


Figure 3.1.5.2 (a) big learning rate. (b) small learning rate

In practical implementations, the learning rate is varied until a compromise is obtained between keeping equation (7) valid, while not making the algorithm too slow.

Example

In this section, an example of handwritten digit recognition, using neural networks will be explained. This idea is elaborated in more details in Michael Nielsen's book, *Neural Networks and Deep Learning* [2].

First of all, an image containing several digits needs to be split into several small images, each containing a digit. For instance, the initial image is segmented, like in Figure 3.1.5.3.



Figure 3.1.5.3 – Sample of MNIST dataset, source: [16]

Choosing the Network architecture

After this step, an algorithm needs to be implemented, which for instance recognizes correctly the digits; this is the real challenge, since a neural network needs to be implemented, in order to

recognize the handwritten digits. A simple way to solve this problem is to implement a neural network with only one hidden layer, as in the figure below:

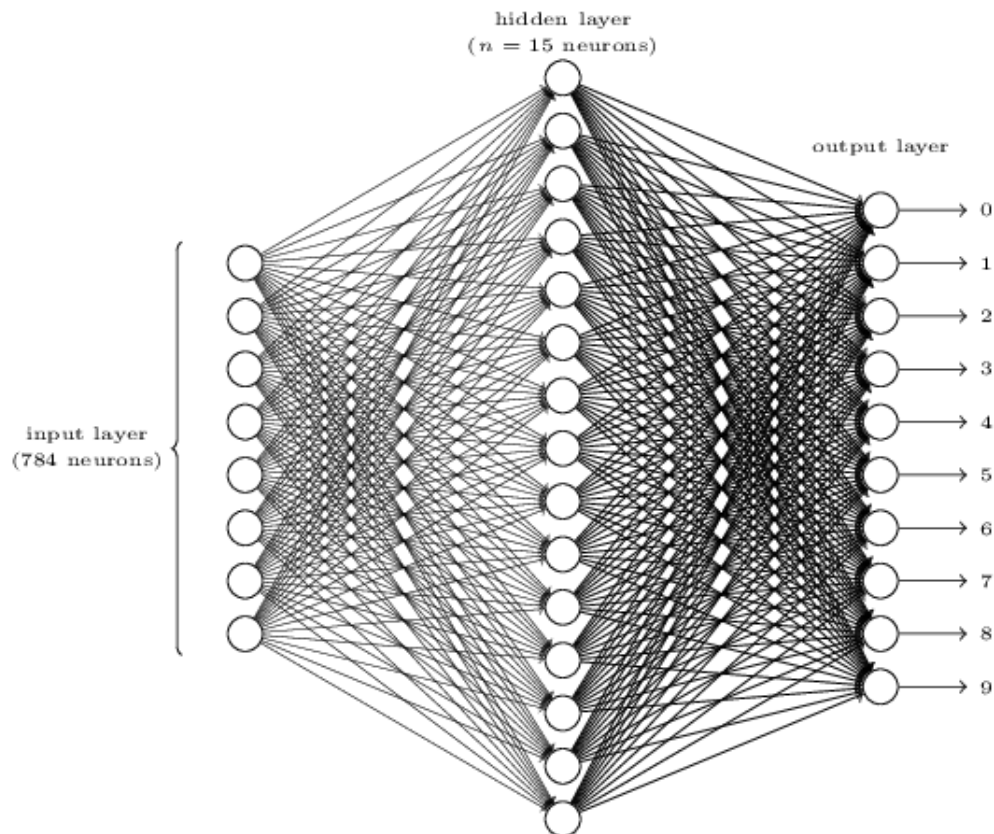


Figure 3.1.5.4 - Neural network with one hidden layer; source: [2]

The layers of this network are analyzed:

- ❖ The input layer contains neurons in which values of the input pixels are encoded; the provided images are 28x28 pixels and they represent black and white scans of handwritten digits. Thus, the input layer needs to have 784 (28 x 28) neurons, each pixel having the values between 0 (representing white) and 1 (representing black);
- ❖ The hidden layer of the network has $n = 15$ hidden neurons; but it can have a higher number as well;
- ❖ The output layer has 10 neurons, each representing the corresponding digit that needs to be decoded by the network. For instance, if the 5th neuron fires, then the output of the network will be 4, because the first neuron indicates the digit 0.

To understand what the hidden layers do, let us consider the example of the 5th output neuron trying to decide if the input image is a “4” or not. To make this decision, the output neuron takes the information provided by the hidden one. This neuron takes parts of the image and compares them to what those particular parts would look like, if in the image the digit “4” was present. For this particular example, the first hidden layer might detect if an image like in Figure 3.1.5.5 (a) is present in the input; one possible way to do this is by assigning higher weights to the input pixels that overlap with the ones from the image and smaller weights for the other ones. Following the same example, it can be supposed that the second, the third and the fourth hidden neurons detect whether the images from Figure 3.1.5.5 (b), (c) and (d) are present in the input image.

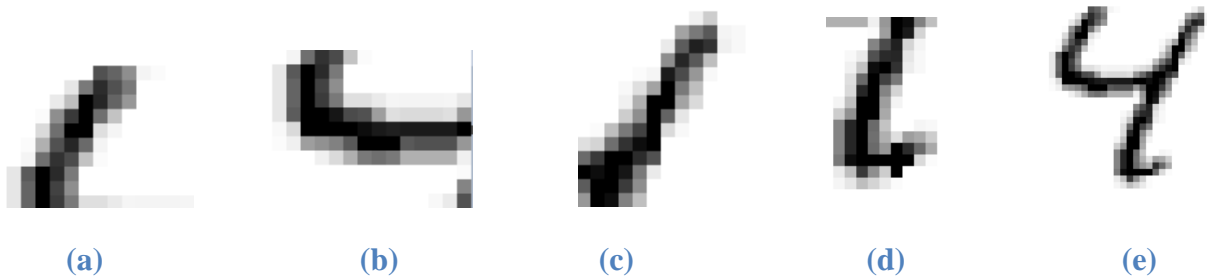


Figure 3.1.5.5 - Small parts of digit “4”

Of course, all of the images together make the initial image of the digit “4”, shown in picture 3.1.5.5 (e); and if all neurons fire, the conclusion is that the input digit was a “4”.

Even though in this presented case, the output is obviously depending on the hidden layers’ output, there are many other cases; the network can operate different than detecting parts of the initial image.

The learning process

Once the architecture is designed, it needs to be learned to recognize digits. This process involves two steps: choosing the dataset to learn and learning with gradient descent.

A good database for training handwritten digits is MNIST - Modified National Institute of Standards and Technology database. This contains more than 70 000 images for training and testing, with the specifications mentioned in the previous section.

The following notations can be made:

- ❖ x – training input
- ❖ $y(x)$ – its corresponding desired output

As mentioned before, the input x is a 784-dimensional vector and each of its entries represents different shades of gray for a single pixel in the image. The output is a 10-D vector; so, for instance, if an image is detected as containing the digit “3”, $y(x) = [0,0,0,1,0,0,0,0,0,0]^T$.

Following, a cost function is defined, as in equation (3); the gradient descent is applied as previously explained and finally, the optimal weights and biases are obtained. Having these parameters, the program can be implemented and tested.

3.2 Convolutional Neural Networks

The CNNs are similar to the usual Neural Network presented before; they comprise of neurons with learnable weights and biases and each neuron receives inputs, performs a product and gives it to its output. Also, the scope of the network is to classify *images* - provided as pixels to the network – and even classify objects, like faces, traffic signs and many others.

The difference is the following: the architecture on ConvNets makes the explicit assumption that the inputs are *images*. This is extremely useful, because it makes the forward function more efficient and also reduces the number of parameters of the network.

Architecture

Regular Neural Networks do not scale well to full images, because the number of weights is huge for a good quality picture; having a fully-connected structure is not a good idea in this case. For instance, for a picture with size $1,280 \times 720 \times 3$ (1,280 horizontal pixels x 720 vertical pixels x 3 color channels), the neurons would need to have $1,280 \cdot 720 \cdot 3 = 2,764,800$ weights. Consequently, this fully-connected architecture is not used for the convolutional neural networks.

A novelty that the CNNs come with is the *3D volume of the neurons*. This means that since ConvNets are used for images as input, a specific feature for their architecture appears: the neurons have 3 dimensions: width, height and depth. Also, the neurons of a layer are connected to a small region of the previous one, and not fully connected. In the figure below, there is a visualization of a regular three-layer Neural Network (Figure 3.2.1 a) and of a ConvNet equivalent (Figure 3.2.1 b).

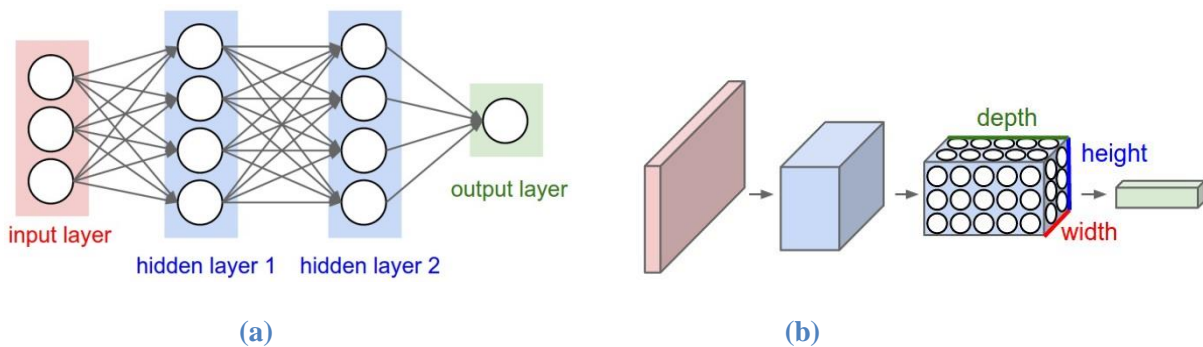


Figure 3.2.1 (a) – Regular Neural Network. (b) Convolutional Neural Network; source: [3]

Layers

As described above, a convolutional network is formed of several layers and each of them transforms one volume of activations into another, through a function. Three types of layers are: the convolutional layer, the pooling layer and the fully connected one. The function of these layers is exemplified below, through an example of image classification, using an image from the CIFAR-10 dataset (Canadian Institute For Advanced Research).

- ❖ Input layer – this is identical with the one from the general neural nets, so it keeps the input image's pixels values; for the analyzed CIFAR image, the dimensions of each layer are given in Table 3.2.1;
- ❖ Convolutional layer (CONV) – computes the output of neurons connected to input (totally or just locally), but computing the dot product between their weights and a small region with which they are connected to input;
- ❖ Rectified Linear Unit (ReLU) Layer – applies an element wise activation function, as shown in the last graph from Figure 3.1.4.2;
- ❖ Pool Layer – performs a down-sampling of the input image, to reduce its dimensions;
- ❖ Fully Connected (FC) Layer – computes the final scores of a class.

Layer	Dimension	Details
INPUT	32X32X3	if the image has: width 32, height 32 and three color channels R,G,B
CONV	32X32X12	if 12 filters are used
RELU	32X32X12	this layer leaves the volume unchanged
POOL	16X16X12	down sampling on width & height only
FC	1X1X10	each of the 10 numbers correspond to a class score

Table 3.2.1 - Layers' dimensions for a CIFAR-10 input image

A convolutional network's simplest architecture consists of the above mentioned types of layers that aim to transform the image volume, as pixels, into an output volume, that holds the class scores.

Each layer transforms a 3D input into a 3D output through a differentiable function. It is interesting to notice that some layers contain parameters and some do not; the CONV and FC layers' transformations depend on the activations in the input volume and also on the weights and biases of the neuron, whereas RELU and POOL implement a fixed function. So, for the first case, the specific parameters are trained using *gradient descent*, in order to make the scores of the class resulting from the CNN correspond with the labels in the training set for each image.

Case Study

In the following paragraphs, an explanation about how a given architecture learns to recognize images will be given. For this, the convolutional neural network from Figure 3.2.1 shall be analyzed. This architecture aims to classify an input image into one of the four categories: dog, cat, boat or bird. As it can be seen from the figure below, when an image with boats is fed at input, the network classifies it correctly, by assigning the highest probability of 94% to that class.

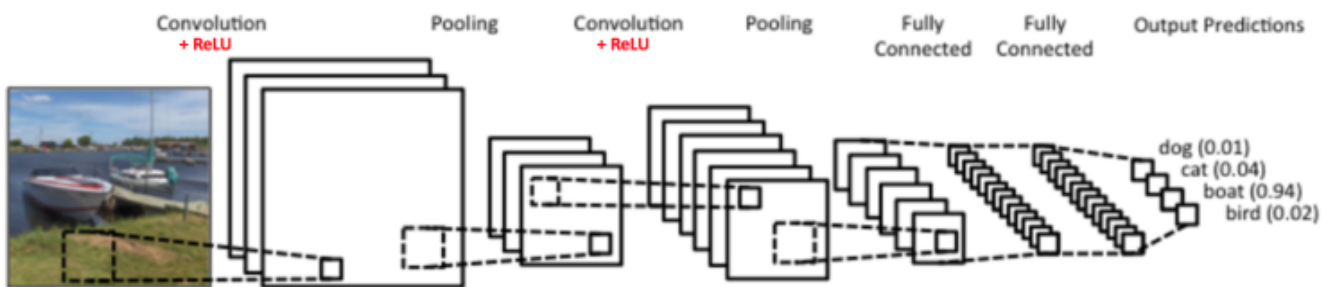


Figure 3.2.1 – An example of ConvNet, source: [3]

In the proposed CNN there are four main operations, which will be explained in what follows.

1. Convolution
2. Non Linearity (ReLU)
3. Pooling (sub-sampling)
4. Classification (fully connected layer)

An image is represented by multiple pixels

To understand how the neural networks process the images, it is necessary to know how the computer perceives an image; and that is by a matrix of triplets, where each triplet represents the

R,G,B values that correspond to each pixel. In the figure below, this idea is illustrated; the corresponding matrix was obtained using the *imread* function in Matlab and one of the three channels is shown.

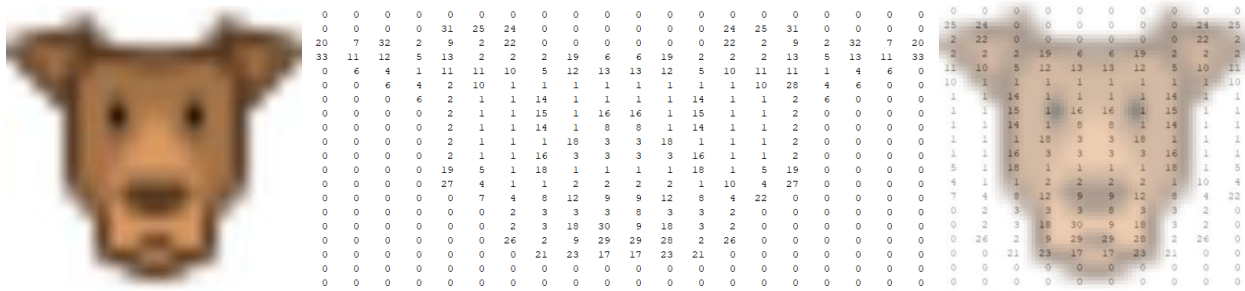


Figure 3.2.2 - The representation of a picture by a matrix of numbers

Intuitively, the better quality of the picture, the bigger the size of the corresponding matrix. So, in order to feed the image of the dog on the left of Figure 3.2.2 to a neural network, the corresponding matrix to its right needs to be fed at input.

A. Convolution

In the case of ConvNets, the convolution operator is meant to extract features from the input image. It does this while preserving the spatial relationship between pixels and this process of learning is done on small parts of the input data.

To understand how this process works, the grayscale image in Figure 3.2.3 is considered, which has as pixel values only 0 and 1, that is a pixel is either black or white.

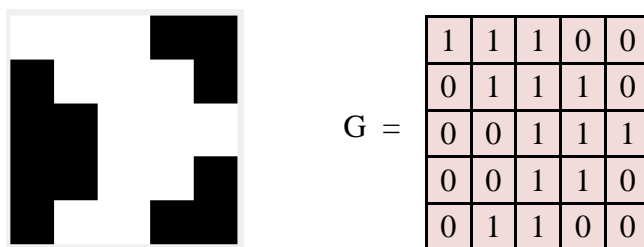


Figure 3.2.3 - The grayscale image to be analyzed and its corresponding matrix, G

Apart from this, the following 3x3 matrix F, called *filter* is considered:

$$F = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The convolution process and its results are presented in Fig. 3.2.4:

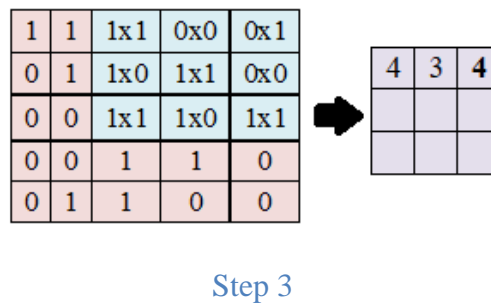
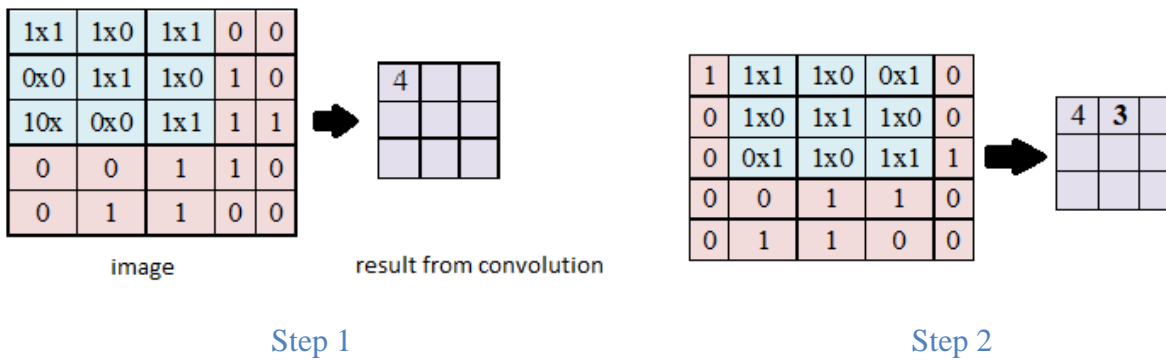


Figure 3.2.4 - The convolution process

And so on, until:

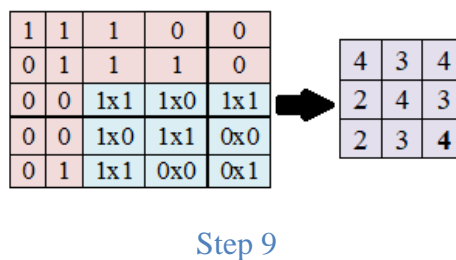


Figure 3.2.4 - The convolution process

So, the convolution process is described as follows: the filter matrix F slides over the input image, represented by G , by one pixel (by one *stride*); for every position, the element-wise multiplication is computed between the two matrices and the results are added. The final number resulting from this sum represents one element of the output matrix.

As terminology, the following terms are used: F is named *filter* or *feature detector* and the resulting matrix from sliding the filter over the original image is called *Convolved Feature* or *Feature Map*.

Obviously, different values of the filter matrix produce different feature maps, for the same image. Thus, operations like blur, sharpen, edge or curve detection can be performed, by a proper choice of the filter's numerical values, depending on the wanted effect.

In practice, during the learning process, a convolutional neural network learns the values that the filters need to have, in order to extract specific parameters. Naturally, more filters in the network mean more extracted features and this leads to better patterns recognition in new images.

The most important parameter that controls the Feature Map is the *depth*. This is the number of filters used in the convolution. In the network presented in Figure 3.2.1, the first convolution operation is made using three different filters, resulting in three feature maps, as shown below.

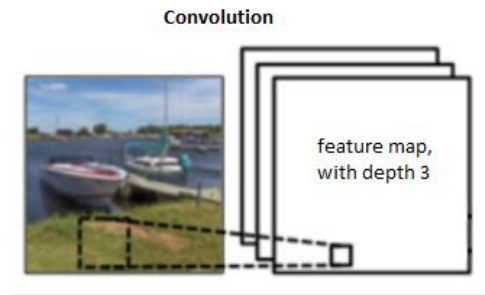


Figure 3.2.5 - Convolution with three filters

B. Non Linearity

After every convolution operation, an additional one was introduced in the network from Figure 3.2.1: an activation function. From the ones presented in Table 3.1.4.1 – Activation functions, the last one was chosen, because it has been found to perform better in most situations.

ReLU is the abbreviation from Rectified Linear Unit; its function is

$$\text{output} = \max(0, \text{input})$$

and its graphical representation is in Figure 3.1.4.2 – Graphical representation of the activation functions; from the latter one, it can be observed the non-linearity of this function.

Most of the real world data that the CNN needs to learn is non-linear, but the convolution is a linear operation, since it is an element-wise multiplication and addition. This is why a non-linear function is applied after a CONV. The output feature map after this operation is called the *Rectified Feature Map*.

C. Pooling

This operation reduces the dimensions of the input representation, while preserving the most important information. Its main functions are:

- ❖ reduces the size of the input feature map
- ❖ reduces the parameters and computations in the network
- ❖ reduces the effect of small input variations on the output; this is because after taking the maximum, or the average value of an area in a matrix, a small distortions in input does not necessarily affect the output
- ❖ helps detection of objects in an image, no matter their location; pooling is useful for making an almost scale invariant representation of the image

Pooling can be of different types, like max, average, sum and others.

The case of Max Pooling will be presented. In the *Rectified Feature Map* presented above (after convolution and ReLU), some spatial neighborhoods are defined – in Figure 3.2.6 they are represented by the four colors - and the largest element is kept.

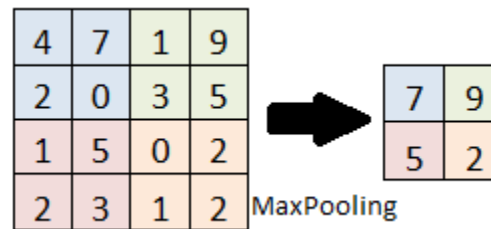


Figure 3.2.6 - Example of MaxPooling

D. Classification

The fully-connected (FC) layer is a general multi-layer perceptron, as described in section 3.1.3 Multi-layer Perceptron; every neuron in this layer is fully connected with the ones from the previous layer.

This last layer is meant to use all of the features provided by CONV and POOL layers, in order to classify the input image into a class. Specifically, for the network given as an example in Figure 3.2.1 – An example of ConvNet, there are 4 possible outputs, as show in Figure 3.2.7.

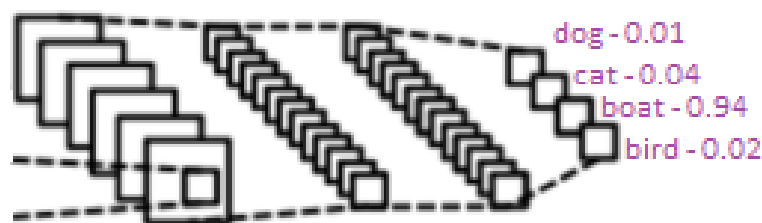


Figure 3.2.7 - The fully connected layers from the network given as example in Figure 3.2.1

The four possible predictions for the input image are accompanied by their corresponding probabilities and the sum of all must be one. This is achieved by using the *Softmax* function; what it does is to transform a vector of scores into one with values between 0 and 1, such that the sum of the elements is one.

Training using back propagation

The training process of the Convolutional Network is briefly the following:

- (a) **Initialization of parameters** - all filters and weights are initialized randomly
- (b) **Forward propagation** – the input is fed and it goes through the network – convolution, ReLU, pooling and finally propagation in the fully connected layer; finally, it finds output probabilities for each class (for the first training example, these are random, since the initialization of the parameters is also random)
- (c) **Error computation** – the total error at the output layers is computed

- (d) **Back-propagation** – through it, the gradients of the error, with respect to the weights are computed; afterwards, using gradient descent, the filter weights are updated, in order to minimize the output error.

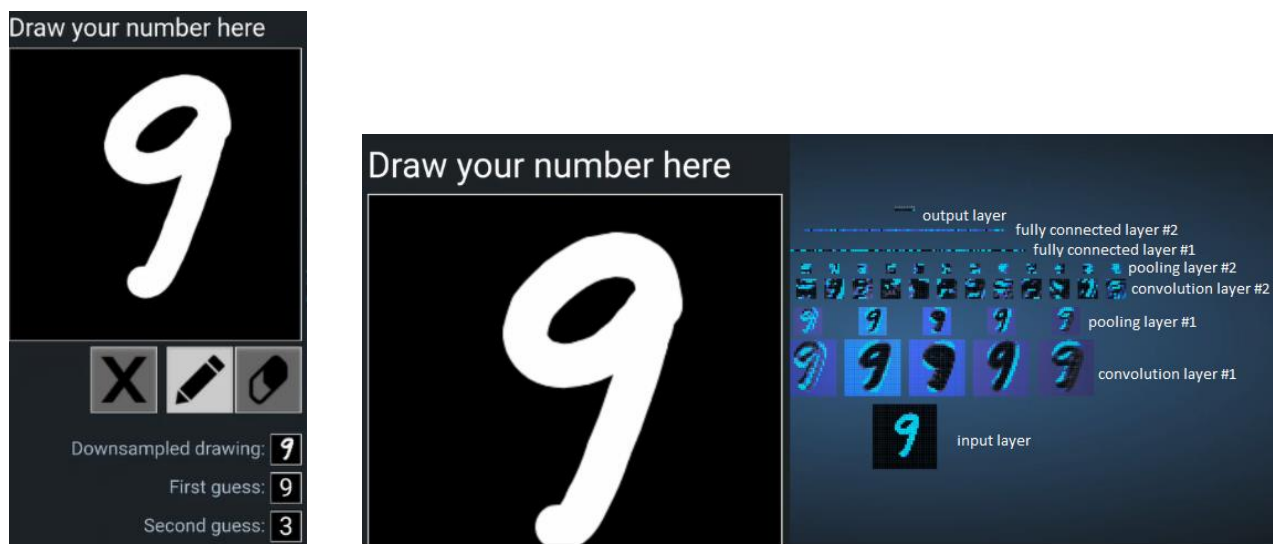
The steps (b), (c) and (d) are repeated for all images in the training set.

In this way, the network learns. For instance, if in the beginning, the network given as example in Figure 3.2.1 had as prediction vector [0.1 0.15 0.5 0.25], when the image is fed again, the weights are adjusted and that vector might improve to [0.05 0.1 0.8 0.05], which is closer to final wanted probabilities, [0 0 1 0].

Example of a Convolutional Neural Network

In order to visualize a ConvNet, an application was created by Adam Harley[4]; it is intuitive and offers a good way to understand how the layers of a ConvNet work.

Firstly, a digit is drawn by the user, as in the picture below.



(a) introducing the input

(b) layers visualization

Figure 3.2.8 - Visualization of a ConvNet's layers

The image is down-sampled to 32x32 pixels (1024) and then the first convolution layer is formed, by convolution of 6 different filters with the input image; from here, as it can be seen in Figure 3.2.8 b, at the **convolution layer #1**, since **six** 5x5 filters are used, a feature map of depth **6** is obtained.

The next step is pooling and a 2x2 MaxPooling is made over the six feature maps previously obtained at the **pooling layer #1**.

Afterwards, these two steps are repeated and next are 3 **fully connected layers**: 120 neurons in the first one, 100 in the second one and 10 in the last FC, which is the output layer. Even though not specifically represented, each node from a FC layer is connected to each one from the previous layer.

The output layer has only one bright node, which corresponds to the digit “9”, as that one has the highest probability among all digits.

Other network architectures

- ❖ **LeNet Architecture** – was one of the first convolutional neural networks; it was used for character recognition, like zip codes or digits. An explanation on how images are recognized using this architecture has been developed in this paragraph, since the main concepts of LeNet Architecture are used in the networks from nowadays.

Network	Dataset	Number of Parameters [millions]	Validation error [%] (top - 1)
VGG-19	ILSVRC-2012	144	24.4
AlexNet	ILSVRC-2010 & ILSVRC-2012	62	37.5
Res-Net101	IMAGENET	44.5	21.8
Inception-v4	ILSVR 2015	65	17.7
GoogLeNet	ILSVRC 2014	11	31.3
MobileNet	IMAGENET-2012	3.3	33.49

Table 3.2.2 - Comparison between different architectures, considering the dataset used for training, the number of parameters and the validation error

- ❖ **AlexNet** – it is a deeper and wider version of the LeNet architecture and won in 2012 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It was a real breakthrough at that moment and current applications of ConvNets can be attributed to this work.
- ❖ **GoogLeNet** – this architecture won the first prize in the same competition in 2014 and the novelty is the development of an Inception Module, which reduced considerably the number of parameters in a network.
- ❖ **VGGNet** – its main contribution was proving that the depth of the network is a critical component for good performance.
- ❖ **ENet** – the real-time applications, that require low latency are made using this architecture developed in 2017, since it is faster and requires less parameters than existing models
- ❖ **SE** – an innovation has been brought in 2018 in this domain by the Squeeze-and-Excitation Networks; the SE block “adaptively recalibrates channel-wise feature responses by explicitly modeling interdependencies between channels”. [5]

As a comparison between all present architectures, there are many factors to take into account, like power consumption, accuracy, number of operations, size of the network, memory used and this is why a comparison between architectures is very difficult. However, Table 3.2.2 presents different architectures, and their corresponding dataset used for training, the number of parameters and the validation error. In this way, a visual analogy can be made; as a remark, the networks are ordered chronologically.

3.3 Mobile Nets

As previously described, since deep learning has enhanced the progress in computer vision, the problem of image detection found its answer with the development of ConvNets; from there, many variations have been designed, in order to assure the requirements for different applications.

Many mobile deep learning applications used to be performed in the cloud; so, when an image was fed for a classification, it was sent to a server and the classification was done remotely, afterwards sending the result to the mobile application.

Using *MobileNet* neural networks [6], this is no longer the case, thus having the advantage of a portable application, without the need of external devices or an Internet connection.

This architecture was designed to effectively maximize the accuracy, while compromising on the limited resources of an embedded application. Apart from its small size, low latency and power, MobileNets has several versions, which offer the possibility of tuning the resource – accuracy trade-off for a specific problem. The release contained the model definition, as well as 16 pre-trained classification checkpoints for use in projects. Using the biggest MobileNets, 1.0, 244, an accuracy of 95.5% can be achieved with just 4 minutes of training, as for the smallest, 0.24, 128, an accuracy of 89.2% can be achieved, using just 930kb of memory [9], depending, of course on the hardware used. As far as the final implementation presented in this paper, using the SSD300 model, trained with MS-COCO dataset, on an NVidia Quadro M4000, the training process lasted 8 days.

In the first section, the architecture, advantages and trade-offs of MobileNet will be presented, while in the second one a SSD model will be explained, together with its advantages. Also, the influence of the SSD model on computational speed and size will be described.

3.3.1 Introduction

Since many research work focus on the development of a small network, rather than optimizing its volume for the speed of the process, the MobileNets represent a great advantage, for they are:

- ❖ low latency models
- ❖ small sized
- ❖ suitable for mobile and embedded applications

While the general trend is to compress a pre-trained network or to train a small one, the priority of MobileNets is to optimize small networks for latency.

3.3.2 Architecture

MobileNets are developed from depth-wise separable convolutions that are used to reduce the computation volume for the first layers.

Depth-wise Separable Convolution

The MobileNet model is based on depth-wise separable convolutions, factorizing the standard convolution operation into 2 sub-operations: a depth-wise convolution and a 1×1 convolution, called a point-wise convolution.

The depth-wise convolution applies a single filter to each input channel, whereas the point-wise convolution applies a 1×1 convolution, to combine the outputs the depth-wise convolution.

Unlike the standard convolution, which filters and combines the outputs in a single step, the depth-wise separable convolution splits this into two layers: one for filtering and another one for combining. The effect is of drastically reducing computation and model size.

The idea of splitting the standard convolution (a) into two separate layers: a depth-wise convolution (b) and a 1×1 point-wise convolution (c) is presented in Figure 3.3.2.1 below.

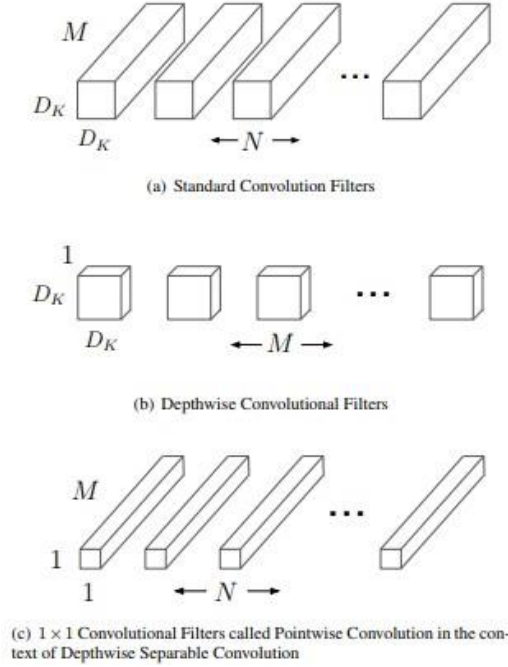


Figure 3.3.2.1 – Splitting standard convolution into depthwise and pointwise convolutions, source: [6]

The input of a standard convolutional layer is a feature map F of dimensions $D_F \cdot D_F \cdot M$ and it produces a feature map G of dimensions $D_G \cdot D_G \cdot N$; where:

- ❖ D_F is the spatial width and height of a square input feature map;
- ❖ M is the number of input channels, so the input depth;
- ❖ D_G is the spatial width and height of a square output feature map;
- ❖ N is the number of output channels, so the output depth.

The convolution kernel K - or the convolution matrix, as previously explained - has the size

$D_K \cdot D_K \cdot M \cdot N$, where D_K is the spatial dimension of the kernel (assumed to be square) and M and N are defined as previously.

The *standard* computational cost of the *convolutions* is: $D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$.

MobileNet uses depth-wise convolutions to apply a single filter on each input channel. Point-wise convolution, a simple 1×1 convolution, is then used to create a linear combination of the output of the depth-wise layer. MobileNets use batchnorm and ReLU nonlinearities for both layers. K is the depth-wise convolutional kernel, of size $D_K \cdot D_K \cdot M$, where the m^{th} filter in K is applied to the m^{th} channel in F , to produce the m^{th} channel of the filtered output feature map G . *Depth-wise convolution* has a computational cost of: $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$.

With this type of convolution, which only filters the input channels, an additional layer of 1×1 convolution is needed in order to generate the new features. The combination of depth-wise convolution and 1×1 (point-wise) convolution is called *depth-wise separable convolution*.

Depth-wise separable convolutions' cost is: $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$.

By expressing convolution as a two-step process, a reduction in computation of: $\frac{1}{N} + \frac{1}{DK^2}$ can be obtained.

In this way, thanks to 3×3 depth-wise separable convolutions, MobileNets aims to reduce the computation volume between 8 to 9 times than the time necessary for a standard convolution, while making only a small reduction in accuracy.

Network Structure and Training

The MobileNet structure is built on depth-wise separable convolutions, as mentioned in the previous section, except for the first layer, which is a full convolution. All layers are followed by a batchnorm [7] and ReLU nonlinearity with the exception of the final fully connected layer, which has no nonlinearity and feeds into a softmax layer for classification.

Counting depth-wise and point-wise convolutions as separate layers, MobileNet has 28 layers. This type of structure has the advantage of making possible its implementation with highly optimized general matrix multiply functions.

MobileNet spends 95% of its computation time in 1×1 convolutions, which also has 75% of the parameters. Nearly all of the additional parameters are in the fully connected layer.

The final implemented MobileNet model presented in the thesis was trained in Caffe on the COCO image set, using a training script initially developed by the Stanford University researchers, which resulted in a Caffe model with 90 single shot detectable object classes. The training set contains over 100 000 images, the training batch size is 16 and the number of iterations is 400 000. I also used a 0.9 momentum value, which is keeping the loss function decay at a reasonable step, so that it will converge at a high enough speed.

3.3.3 Advantages

As previously mentioned, the advantages of MobileNet are the size and especially the speed, because small neural networks are not generally optimized for higher speed, the developers only taking into account the size of the network. Also, it is important to mention that the MobileNet can be shrunk even more, with the help of 2 hyper-parameters:

- ❖ **α - width multiplier** – its role is to thin a network uniformly at each layer; the number of input channels M becomes αM and the number of output channels N becomes αN . The computational cost of a depth-wise separable convolution with width multiplier α is:

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$
- ❖ **ρ – resolution multiplier** - this is second hyper-parameter that reduces the computational cost of a neural network; it is applied to the input image, so the internal representation of every layer is subsequently reduced by ρ .

The computational cost for the core layers of the network, as depth-wise separable convolutions, with width multiplier α and resolution multiplier ρ becomes:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F,$$

where $\rho \in (0, 1]$ and $\alpha \in (0, 1]$. These parameters reduce the computational cost by ρ^2 , respectively α^2 .

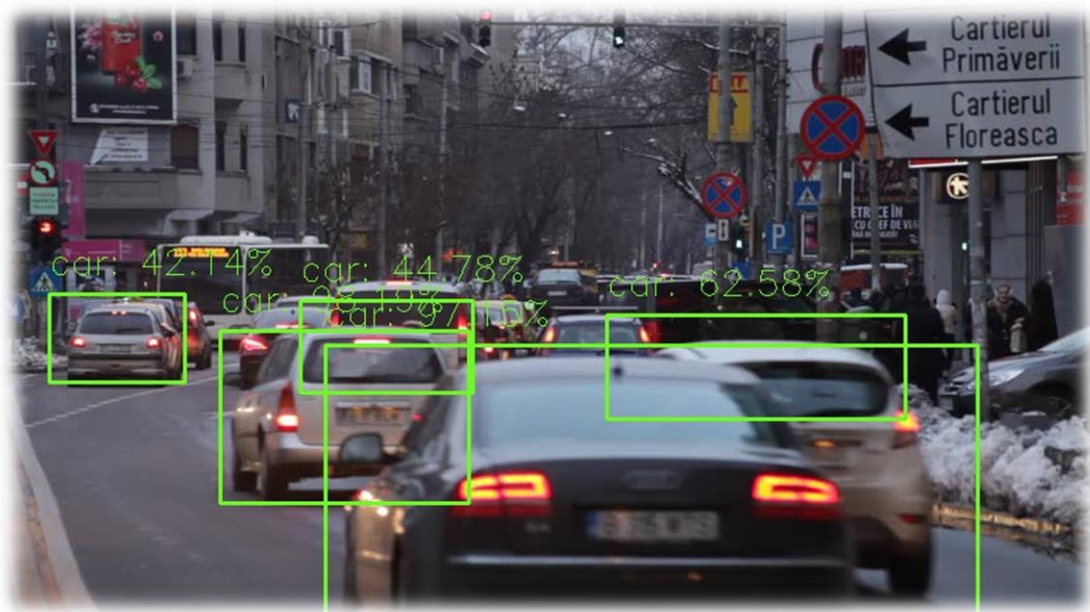
In conclusion, the cost is much lower than the cost of using a full convolutional layer and the neural network has a much smaller size and it is scalable. Even more, with the α and ρ parameters, these optimizations result in a small cost of the accuracy, depending on the hyper-parameters that are chosen. With the hyper-parameters set on 1 (this representing the standard MobileNet) the accuracy will decrease with 1% on a MobileNet, in comparison with a deep neural network with full convolutional layers. Rescaling the neural network with the 2 hyper-parameters will slowly decrease accuracy. From the value of 0.25 of the hyper-parameters, the accuracy will drastically drop, so it is not recommended to reach such a value if high accuracy is wanted.

3.4 Single Shot MultiBox Detector

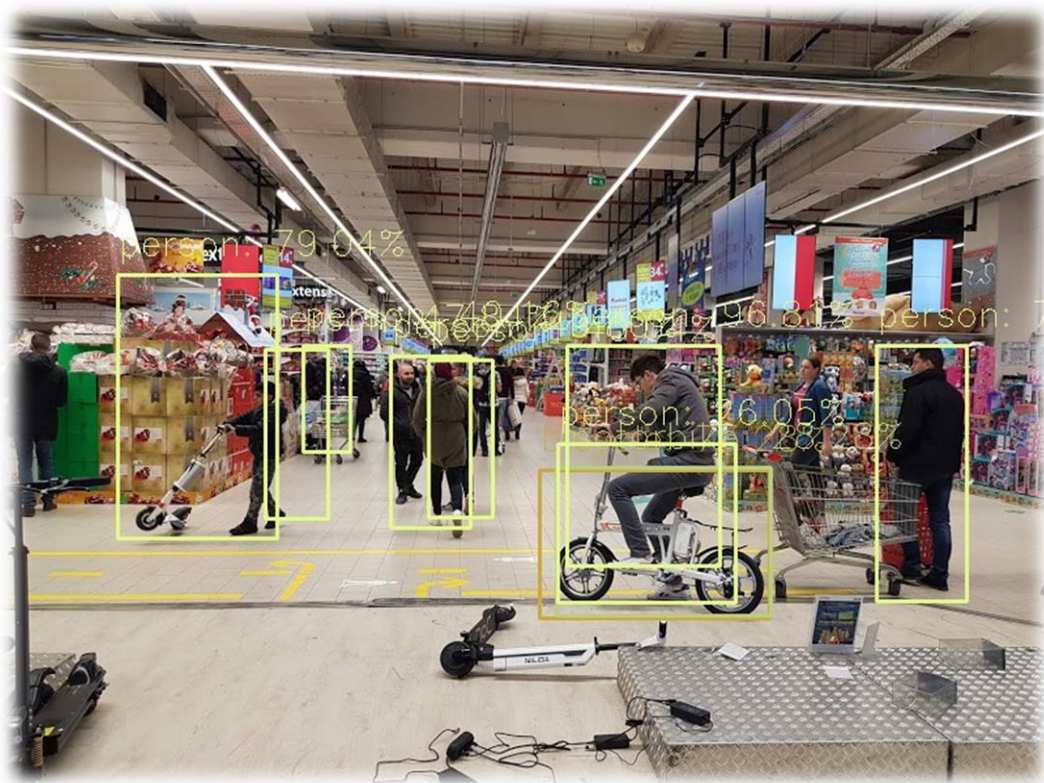
3.4.1 Introduction

The real time detection is achieved using Single Shot MultiBox Detector [8] framework. The novelty consists of the localization and classification being done in a single forward pass of the network.

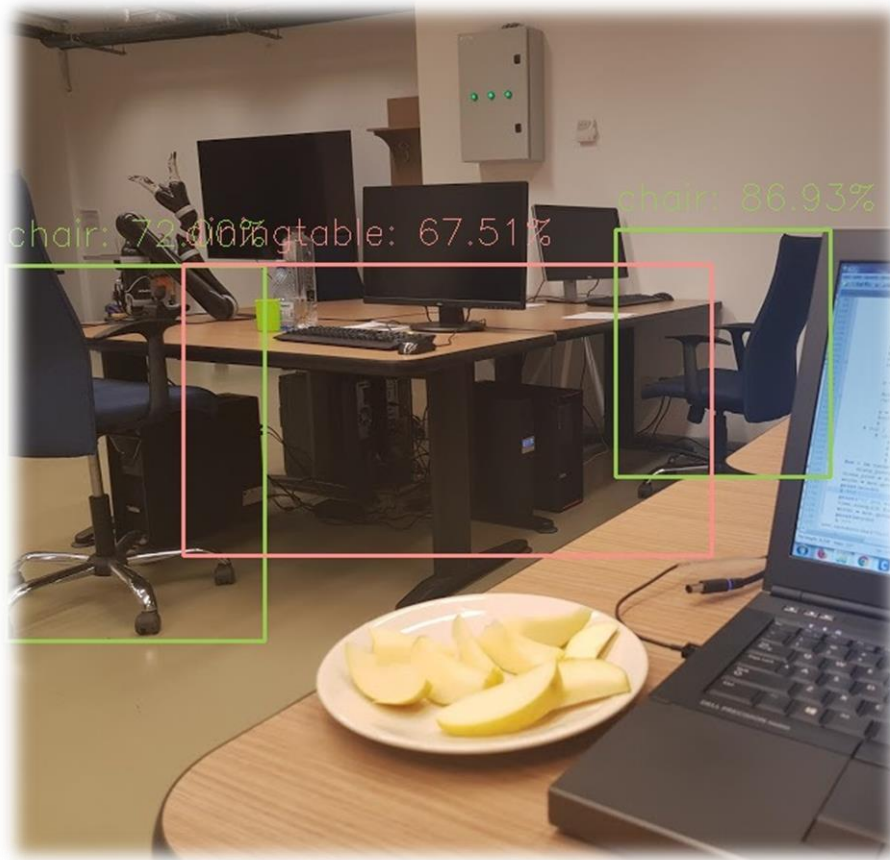
A Single Shot MultiBox Detector (SSD), as described by its authors in [8] is a method for detecting objects in images using a single deep neural network. This method discretizes the output space of bounding boxes into a set of default boxes, over different aspect ratios and scales per feature map location. At prediction time, the network generates scores for the presence of each object category in each default box and produces adjustments to the box, to better match the object shape. Additionally, the network combines predictions from multiple feature maps, with different resolutions, to naturally handle objects of various sizes. Some examples of objects detected using this method are presented in Figure 3.4.1 a, b and c below.



(a)



(b)



(c)

Figure 3.4.1 - Detection examples of SSD on personal dataset

In the above detections, each color corresponds to a category of objects.

3.4.2 Architecture

To understand better what the Single Shot MultiBox Detector does, the provenience of its name should be analyzed:

- ❖ Single Shot – the object localization and identification are done in a single forward pass through the network
- ❖ MultiBox – this is a novelty with regard to the bounding box regression and it is explained in detail in the following
- ❖ Detector – since the network is an object detector

The SSD's architecture is built on the VGG-16 architecture, discarding the fully connected layers. The reason why VGG-16 was used as a base network was because of its great performance in high quality image classification. Apart from this, the FC layers were discarded, adding instead convolutional layers, in order to allow extraction of features at multiple scales, while also decreasing the size of the analyzed frame at each layer.

3.4.3 MultiBox

In this new approach, in order to express the loss of the prediction, two new components were added: *confidence loss* and *location loss*; the first one measures how confident the network is about the objectness of the computed bounding box, while the latter one computes how far the network's predicted bounding boxes are from the ones in the training set. Briefly:

$$\text{multibox_loss} = \text{confidence_loss} + \alpha \cdot \text{location_loss} \quad (11)$$

Multibox Priors

In MultiBox, the '*prior*' entities are created, which are pre-computed bounding-boxes that match as much as possible the original truth boxes. These are used as predictions and the scope is to get them closer to the real boxes. The SSD architecture associates each feature map with default, carefully chosen bounding boxes, hence it generalizes any type of input, without the need of pre-training.

While the accuracy increases, these architectures offer amazing opportunities for real-time detection, from military applications, recognizing traffic signs or lane detection algorithms, to teaching robots like Nao how to interact with the world.

3.4.4 Intersection over Union ratio

This is an evaluation metric that measures the accuracy of an object detector on a particular dataset. The priors discussed above need to be chosen in such a way that their *IoU - Intersection over Union ratio* – is higher than 0.5. Such a number is not good enough, as it can be noticed from Figure 3.4.4.1, but it offers a good starting point for the bounding box regression algorithm and it is certainly better than the previous approach, of starting the prediction with randomly initialized coordinates.

In order to apply the IoU and evaluate an object detector, two boxes are needed:

- ❖ the *ground-truth bounding box* – that is the manually labeled bounding box, in which the coordinates of the object are given
- ❖ the *predicted bounding boxes* – from the model that is tested

The intersection over Union is defined as:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}} \quad (12)$$

This ratio is computed because in practice, it is desired that the two boxes overlap as much as possible. Even though their perfect match is not sought, a higher ratio however indicates a better prediction.

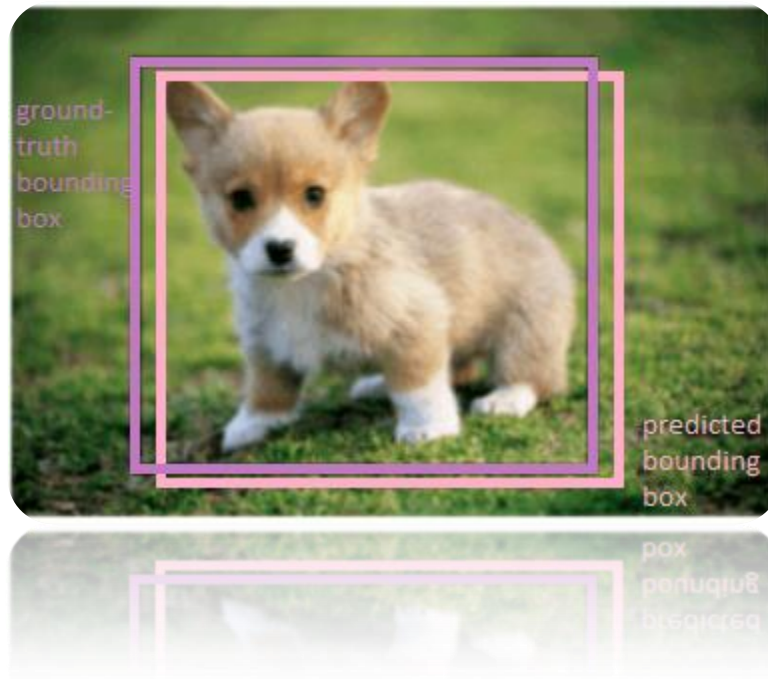


Figure 3.4.4.1 Dog detection example - the corresponding ground-truth and predicted bounding boxes

So, the MultiBox starts with priors as predictions and attempt to regress closer to the ground truth bounding boxes; the result is that this architecture has a total of 1420 priors per image, allowing coverage of input images of different dimensions. In the end, MultiBox retains only a number “K” of top predictions that minimized both the location and confidence losses.

As a remark, the classification is not performed by MultiBox, but by SSD. Thus, for each predicted grounded box, a set of “c” class predictions are computed, for every possible class in the dataset.

In conclusion, combining the MobileNet neural network with a SSD, not only can we obtain a smaller neural network, but also, a faster one, with the cost of a small reduction of accuracy.

Other remarks from the SSD paper are the following:

- ❖ SSD confuses objects from similar categories; this might happen because locations for multiple classes are shared;
- ❖ more default boxes bring forth more accurate detections, with a speed trade-off;
- ❖ having MultiBox on multiple layers result in having a better detection also, since the detector runs on features at multiple resolutions;
- ❖ smaller objects are detected with less accuracy, as they might not appear across all feature maps.

4

THE IMPLEMENTED VISION ALGORITHM

Having in mind all the considered aspects, regarding the limitations of the robot (discussed in Chapter 1, section 1.2 The robot Nao – Technical details and also in Chapter 2, section 2.4. Limitations), a computer vision algorithm was thought, that would have the power to generalize the Aldebaran module, taking into account Nao's limitations; thus, a more complex program was implemented, in Python.

4.1 Working principle

The general flow diagram of the implemented algorithm is presented in Figure 4.1.4 and the working principle is explained in detail in the following.

- ❖ First of all, the arguments that will be given as input for a specific frame are created; they are necessary for the program to know different parameters; these are:
 - ❖ *image* – the path of the capture that will be fed to the network
 - ❖ *model* - the parameters of the network
 - ❖ *confidence* – minimum probability to filter weak detections
- ❖ Afterwards, some presetting for using the *ALSpeechRecognition* module need to be made: the language and the vocabulary are set: the language is English and the recognized words will be “start” and “finish”. More details about this module can be found in the robot's documentation [1].
- ❖ Once this module is on (that is the robot subscribes to the event of SpeechRecognition), if a speaker is heard, the element of the list that best matches what is heard by Nao is placed in the *WordRecognized* key. This will be used in some next steps.
- ❖ Until this moment, the robot is set for speech recognition, having as vocabulary the words "start" and "finish". Therefore, it starts listening for voice commands.

- ❖ While the user's words differ from "start" and the accuracy probability is smaller than a threshold of 0.45, Nao keeps waiting for a valid start command.
- ❖ Considering the start voice command has been given and the robot has a confidence of over 45% of that, the next step is to take pictures using the robot's camera, with the help of another module, *ALPhotoCapture*. The parameters of the capture are presented below.

Camera ID:	top camera
Color space:	RGB
Resolution	VGA (640x480)
Picture format:	jpg

Table 4.1.1 - Parameters of *ALPhotoCapture* module

- ❖ The classes that will be recognized by the convolutional neural network on the robot are stored in a list; the categories of objects that Nao will be able to recognize are the following: plane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow, dining table, dog, horse, motorbike, person, potted plant, sheep, sofa, train and TV monitor. In figures 4.1.1 and 4.1.2, examples of the robot recognizing some of these objects are shown.

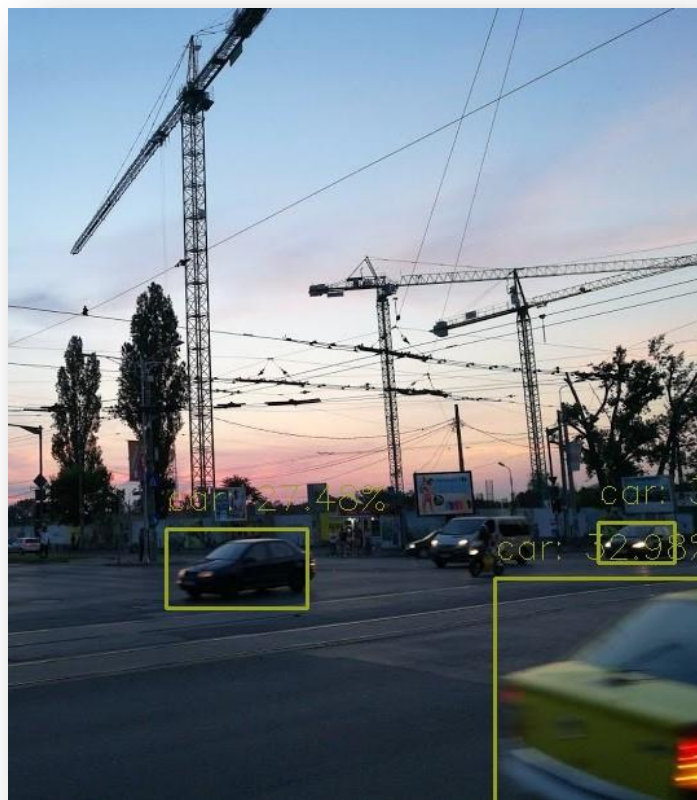


Figure 4.1.1 – Detection algorithm on Nao, with pictures taken with a 12 MP camera (on smartphone)

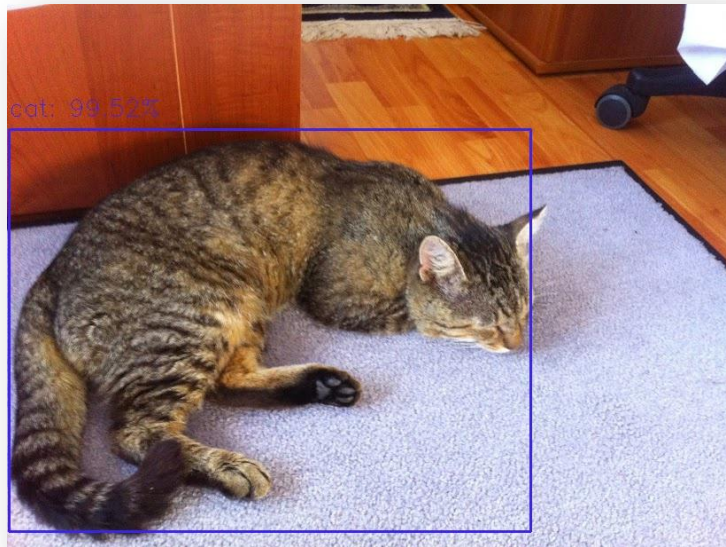
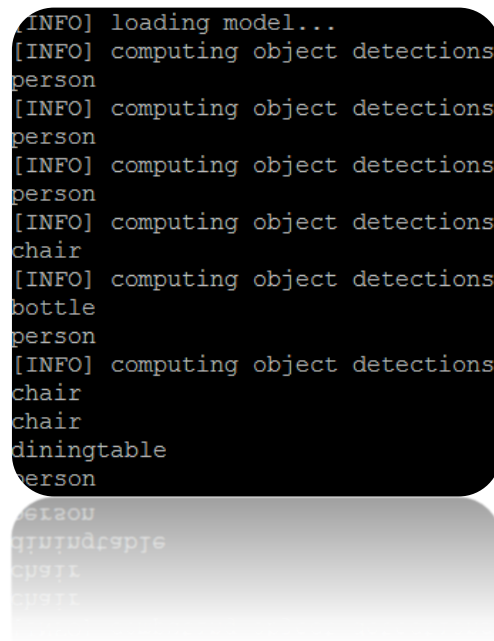


Figure 4.1.2 – Detection algorithm on Nao, with pictures taken with an 8 MP camera (on smartphone)

- ❖ After all the initializations are done and the start command is given, the robot enters a state in which it listens for the stop command, while executing the main program.
- ❖ The next step is to take pictures, with the parameters specified in Table 4.1.1 and to store them into a specific location in Nao's memory.
- ❖ The captured frame is then loaded into the program and it will be used to feed-forward the convolutional neural network; but some preprocessing needs to be made before.
- ❖ This preprocessing is made by the OpenCV's function *blobFromImage*. Briefly, this function creates a 4-dimensional *blob* from the original image; that *blob* contains the necessary features of the image for the network. If necessary, the function *blobFromImage* resizes the image, subtracts the mean values, scales by a factor and swaps the Red and Blue channels. The *blob* is discussed in more details in the section below entitled *Blob*.
- ❖ After the blob is passed through the network, the detections and predictions are obtained for each object in the image, by using the blob as input for the network
- ❖ Following, the weak detections need to be filtered out, by setting a default threshold for the confidence for each detection; in this paper, the threshold was set to 30%. In case of success, the robot outputs the name of the class through the microphone and also on the console, as in Picture 4.1.3.
- ❖ At this moment, the detections for each object are made and ready to be output by the robot, using the *ALTextToSpeech* module.
- ❖ During the tests, some improvements for this initial approach were made.
- ❖ One idea was to prevent the robot from repeating the same class name several times for a single capture. That is, after this refinement, Nao would only say the name of a class once, even though multiple instances could be present in a capture. This solution was brought because repeating the class names is not necessary for the proposed application; its scope is to detect and recognize objects around.
- ❖ Another improvement that was made in order to reduce redundant information was to eliminate the class labels from a capture that were identical to the previous one. In this way,

Nao interprets only the new objects that appear in the frame. A capture before applying this update is show below.



```
[INFO] loading model...
[INFO] computing object detections.
person
[INFO] computing object detections.
person
[INFO] computing object detections.
person
[INFO] computing object detections.
chair
[INFO] computing object detections.
bottle
person
[INFO] computing object detections.
chair
chair
diningtable
person
```

Figure 4.1.3 - The recognized classes by Nao

- ❖ The robot outputs the class names of the detected objects and after this, it has two options: either it stops, due to the voice command, using the same principle explained for the start command, or it continues to capture pictures of the environment and to give information about it.

Blob

A *Caffe* network is composed of layers and each one of them is made up of *blobs*. Therefore, it can be assumed that a *blob* is the basic building block in *Caffe* networks. These entities can be thought of as envelopes, to conveniently access data, since it is of utmost importance to encapsulate data similarly, thus ensuring modularity when designing the networks.

The blobs are 4-dimensional arrays with convenient methods, which store weights of the neurons, activation values and the derivatives of the two. They keep the same dimensions (heights and weight) and the same depth (number of channels), so they will all be processed in the same manner.

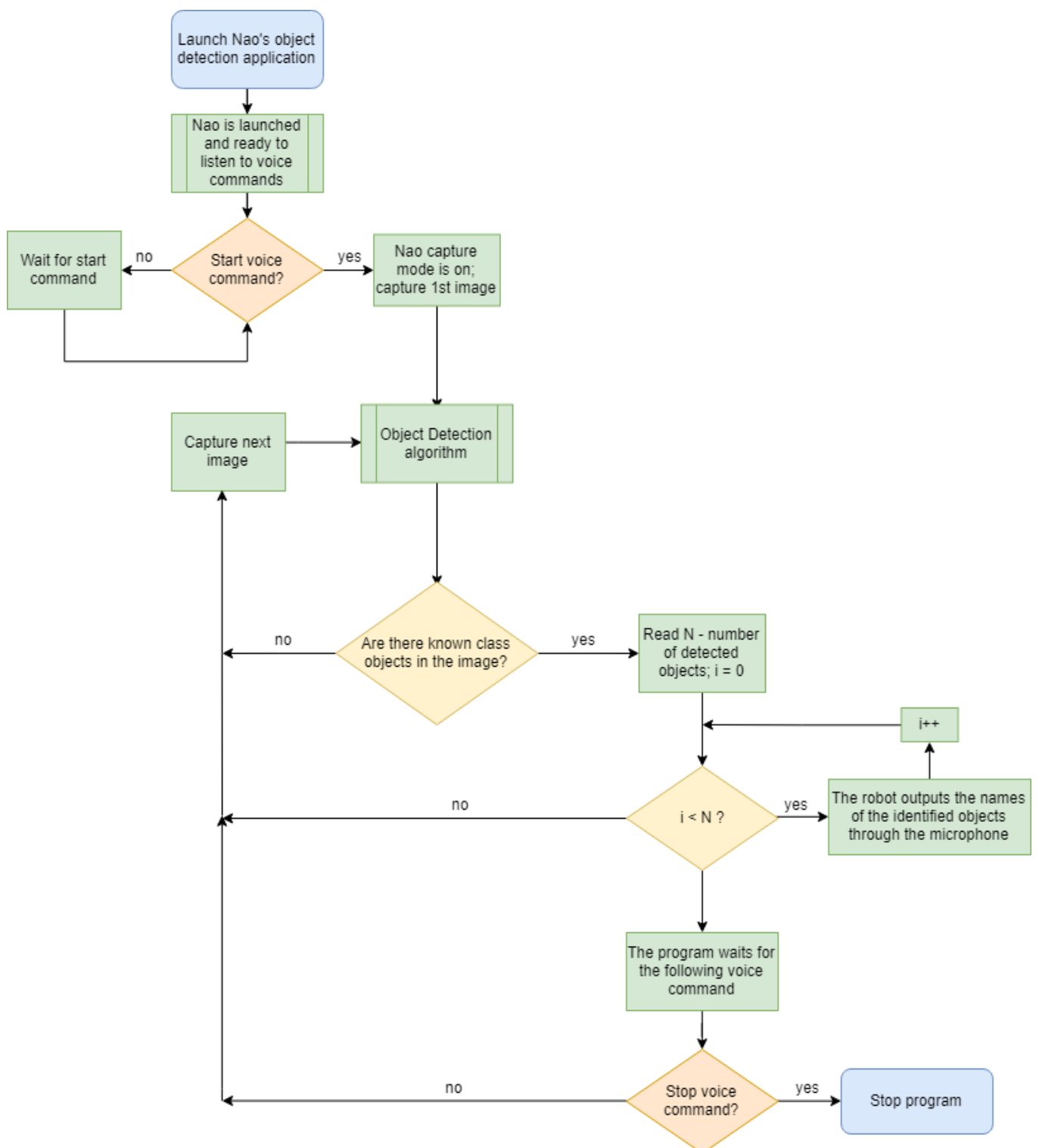


Figure 4.1.4 - General flow diagram of the program

4.2 Object detection using MobileNet SSD

In order to implement on Nao an object detection algorithm, several steps were made. First of all, some algorithms were implemented on a Desktop AMBUJ97, with 8 GB of RAM and a i5-4200H CPU @2.80GHz processor. Afterwards, I wanted to port the successful algorithms on the robot, but because of its limitations, this was a difficult step.

Two main algorithms were tested on the desktop computer and will be presented in the following sections; the first one could only be implemented on the computer and the second one was ported on the robot.

4.2.1 Real time object recognition model, using Tensorflow

The first algorithms that were studied were object classifiers, object identification, shape detection [10], pattern recognition and eventually a more complex algorithm was chosen as a possible implementation on Nao. That was an object recognition program, which could detect and label the objects in a video in real time; this project was achievable, because Google released in June 2017 an Object Detection API (Application Programming Interface), which includes trainable detection models, listed in Table 4.2.1.1 and presented in more details on their blog [11]. The weights were trained on the COCO dataset for each model and one of them is designed to operate on less complex machines, thus it can be run in real time on mobile devices.

About APIs

Briefly, an application programming interface is an application software, which allows two applications to interact with each other; it can be regarded as the messenger that delivers the user's request to the system and in the end providing the results back.

The API that Google released was trained on COCO dataset [12], so on approximately 300,000 images, resulting in 90 detectable classes. Some examples of the trained images are presented below, in Figure 4.2.1.1 a and b.

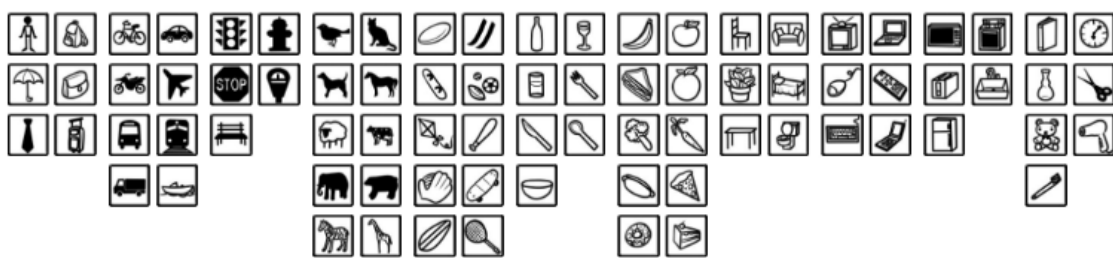


Figure 4.2.1.1 (a) - COCO 2017 object categories, labels not shown; source: [12]



Figure 4.2.1.1 (b) - COCO dataset examples; source: [12]

Models

The API comes with five models, presented in Table 4.2.1.1, each providing a different accuracy and speed when placing the bounding boxes, so that the user can chose the most suitable one for a specific application.

Model Name	Mean Average Precision	Speed
SSD with MobileNet	21	fast
SSD with Inception V2	24	fast
R-FCN with Resnet 101	30	medium
Faster RCNN with Resnet 101	32	medium
Faster RCNN with Inception Resnet v2	37	slow

Table 4.2.1.1 - Precision and Accuracy for the models provided by Google's API

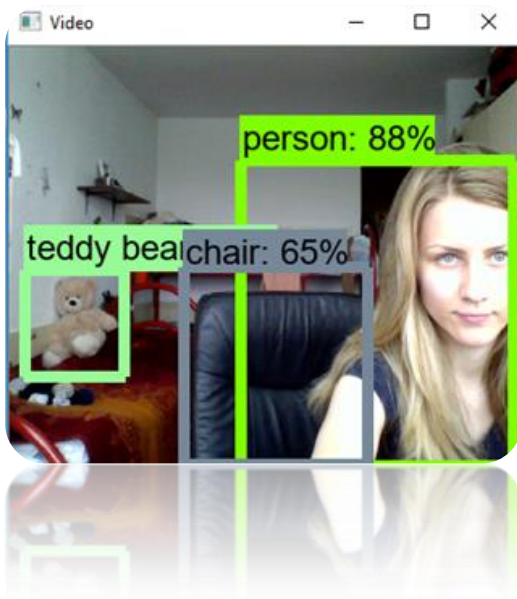
In this table, the notations below were made; they all represent object detection systems that combine different features computed by a convolutional neural network.

- ❖ SSD – Single Shot Detection
- ❖ R-FCN - Region-Based Fully Convolutional Networks
- ❖ R-CNN: Region-based Convolutional Neural Networks

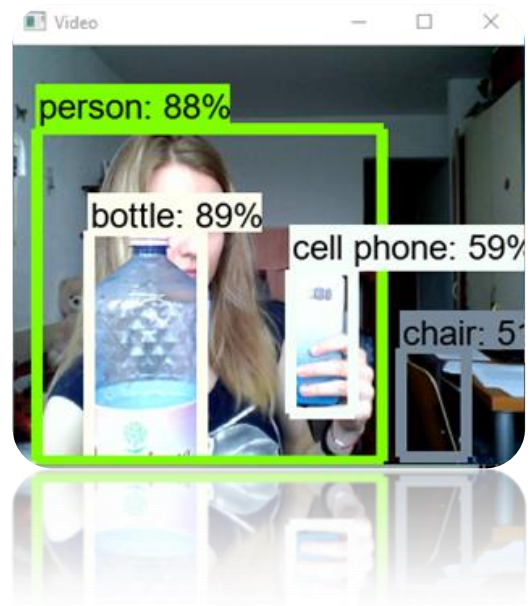
The *mean average precision* (mAP) measures the accuracy of the detection; in this case, it is the product between precision and recall, on detecting bounding boxes. And since the precision measures the accuracy of prediction and the recall is a measure of how good all the positives were found, it can be concluded that the mAP is a good measure of how sensitive the network is, while avoiding false alarms. More details about the models can be found on the git *Tensorflow detection model zoo* [13].

Results

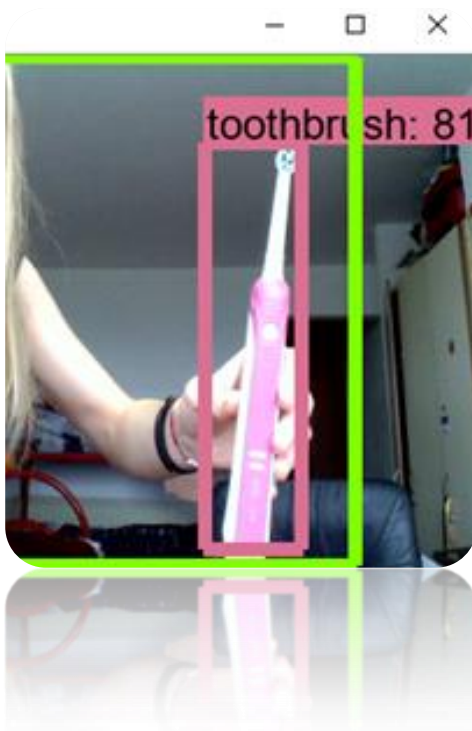
For the practical implementation, the first model from the five was chosen, because it is the most lightweight from all. That model was tested on some real time videos on the computer and the results are shown below.



(a)



(b)



(c)



(d)

Figure 4.2.1.2 – Results from real time detection



(e)

Figure 4.2.1.2 – Results from real time detection



(a)



(b)

Figure 4.2.1.3 – Unsuccessful results from real time detection

Software resources

❖ Tensorflow

Tensorflow is a software library used for numerical computations using flow graphs, in the following way: the mathematical operations are represented by nodes and the edges of the graph stand for data arrays (which are multidimensional), called tensors. These tensors stand between the nodes of the graph.

It was firstly developed in 2015 by the Google Brain team, for internal use at Google in machine learning and deep neural network research area; but today, it is an open source library for numerical computation used in many domains.

One big advantage of Tensorflow is that it has a flexible architecture, allowing the use of more CPUs or GPUs in a desktop or mobile device, without rewriting the code.

However, the biggest problem I encountered regards memory problems. Even though on the computer, this did not appear as a problem, when porting the program on the robot, it turned out to be a huge obstacle, thus leading me to choose an architecture with another model, a Caffe model.

❖ OpenCV

Open Source Computer Vision Library (OpenCV) [14] is another open source library, which was developed for computational efficiency, considering real time applications. It has C++, Java and Python interfaces and supports Windows, MacOS, Linux, iOS and Android.

This library has more than 2500 optimized algorithms, used for several vision algorithms; some examples are listed:

- ❖ face detection and recognition
- ❖ object identification
- ❖ classification of people's actions
- ❖ camera movement tracking
- ❖ moving objects tracking
- ❖ extracting 3D models of objects
- ❖ combining images to get one image of the entire scene
- ❖ finding similar images in a database



source: [14]

Future steps

As it can be observed from Pictures 4.2.1.3 a and b, there are cases when the detection could have been better. For instance, in the first capture from the two, the pear was misidentified, while in the second, the apple was not identified at all. So, there is room for progress for the analyzed model. However, getting closer to the camera solved the last problem, as it can be noticed in Figure 4.2.1.2 d.

Another good improvement would be to speed up the models, in order to integrate them on mobile devices, such as on smartphones or cars. Additionally, other models can be trained with a personal dataset, for personalized applications.

Last, but not least, the biggest upgrade that needs to be made is making Tensorflow less prone to memory problems; improving memory usage and management is an area where Tensorflow developers are working at the moment.

4.2.2 Object recognition model, using Caffe

Considering the memory restrictions that Nao imposes, a smaller sized model, designed using Caffe framework was implemented.

Software resources

❖ Caffe

Another deep learning framework is Convolutional Architecture for Fast Feature Embedding (Caffe); as its developers from Berkeley AI Research team state, it is made with speed and modularity in mind and this is why the final implementation of this project is built on this framework.

One of the reasons why Caffe is used by research and industry developers is its speed; because of this, state-of-the-art models are already implemented on this framework.

Caffe is written in C, with a Python interface; it supports Windows, Linux and MacOS and since its release in April 2017, it has been a powerful tool in vision, speech and multimedia applications.

❖ OpenCV

In August 2017, the version 3.3 of OpenCV was released, featuring an improved deep learning module, *dnn*. This was highly convenient, since it supported Caffe framework.

As a remark, the models are not trained using OpenCV; using it, some pretrained models are taken using deep learning libraries (like *dnn*) and used in programs. Thus, using the version 3.3 from OpenCV, models can be trained on some device – this process being the most time and computational expensive – and afterwards, the model can be transferred on another machine and can feed forward the network, in order to provide an output for the desired application.

❖ Single Shot Detector & Mobile Nets

As previously explained, when these two methods are combined, they provide a very fast, real-time object detection on resource limited devices, like smartphones or development boards.

Final program

The model on which the program was designed is a Caffe version of the Tensorflow implementation of Google's *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* [15]. The model was trained on COCO dataset and afterwards fine-tuned on PASCAL VOC. The fine tuning was made, because the neural network was trained on a large set of images and the final weights were adjusted using a smaller dataset, different from the training one. Eventually, the model could detect 20 objects in images, with a 72.7 % mean average precision. Since the model was trained on COCO dataset - Common Objects in Context - the classes that are recognized are the following: birds, cats, dogs, cows, horses, sheeps, people, bottles, airplanes, bikes, boats, buses, trains, chairs, sofas, tables, monitors and plants.

Implementation on Computer

Below, there are some examples of the previously described program, implemented on the computer.

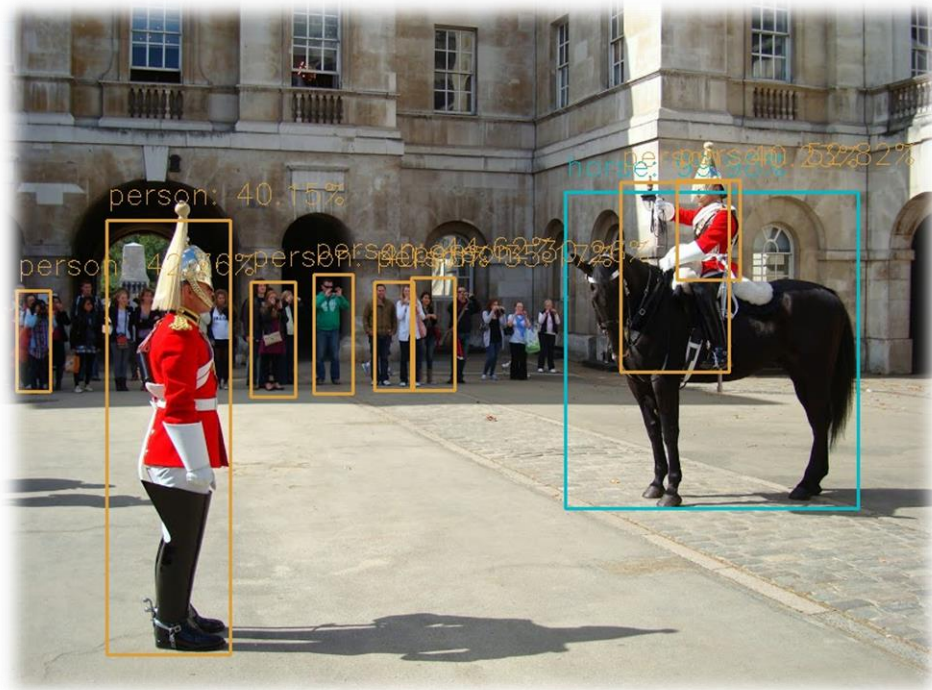


Figure 4.2.2.1 (a) - Successful *person* and *horse* recognition (computer implementation)

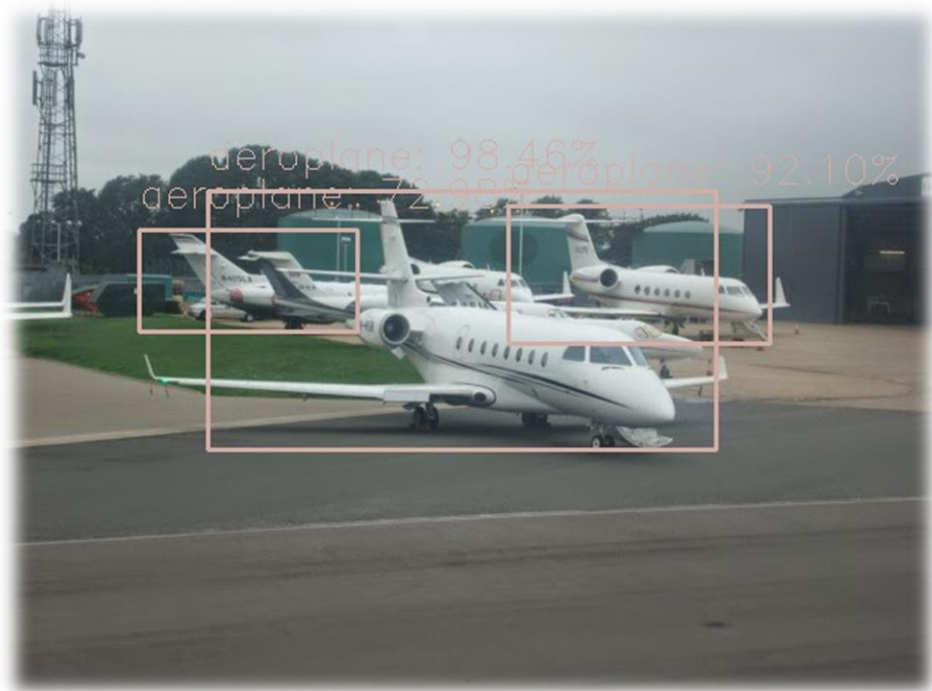


Figure 4.2.2.1 (b) - Successful *plane* recognition



Figure 4.2.2.1 (c) - Successful *car* and *bus* recognition

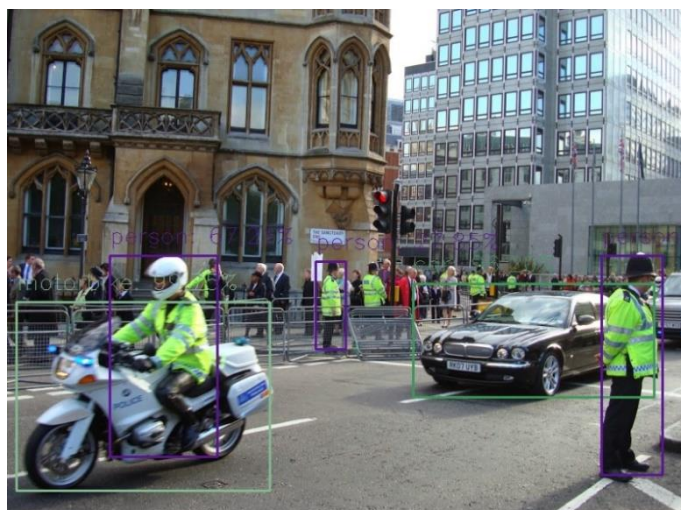


Figure 4.2.2.1 (c), (d) - Successful *person*, *motorbike* and *car* recognition



Figure 4.2.2.1 (e) - Unsuccessful *horse* recognition

Implementation on Nao

After eliminating the memory issues caused by a Tensorflow model, the biggest problem I encountered when importing the program on Nao was when porting the packages. For instance, the *dnn* module is included in a newer version of the cv2 than the robot has pre-installed and porting this package on the proprietary platform of Nao was done by using a cross-compilation tool chain.

After the successful transfer, the robot could be used to capture frames and to feed forward them to the neural network, getting as a result the class in the console. For here on, the improvements presented in section 4.1 Working principle were made.



Figure 4.2.2.1 (f) - Successful *person* and *plant* recognition

In Figure 4.2.2.1 f, an example of the object recognition with the model used by Nao can be seen, with the picture taken in advance and in CHAPTER 5 Experimental Results, more cases are presented, with pictures taken directly from Nao.

As a remark, apart from the grooms, the person behind them was also recognized and considering his position and exposure in the picture, the result is surprising.

5

EXPERIMENTAL RESULTS

The implemented program on Nao had identical results with the computer implementation, since the same network was used. Following, some practical results will be presented, with frames captured by the robot and then, an analysis of the used network will be made.

5.1. Results on Nao

For this part, the object recognition model, using Caffe, described in the previous section was applied on frames taken directly from the robot and the practical results were the following:

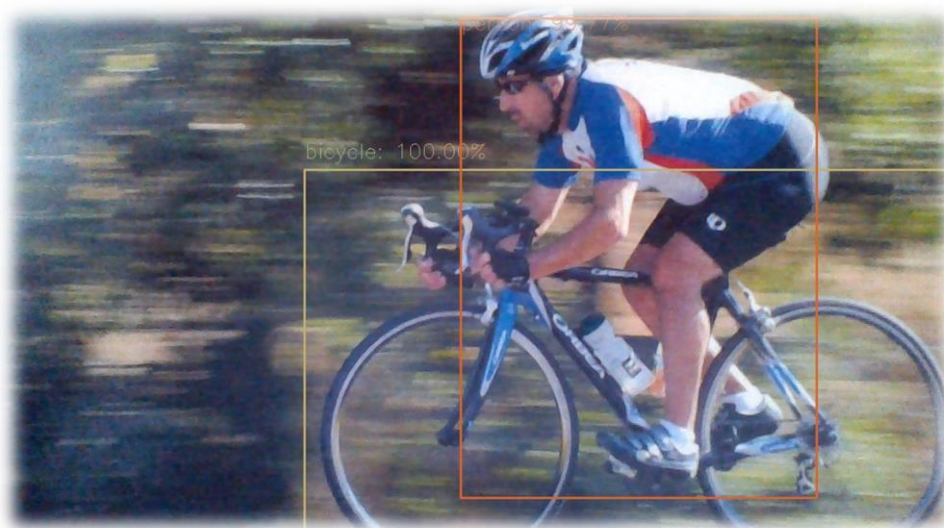


Figure 5.1.1 (a) – Succesful *person* recognition on Nao



Figure 5.1.1 (b) – Successful *bird* recognition on Nao



Figure 5.1.1 (c) – Successful *dog* recognition on Nao

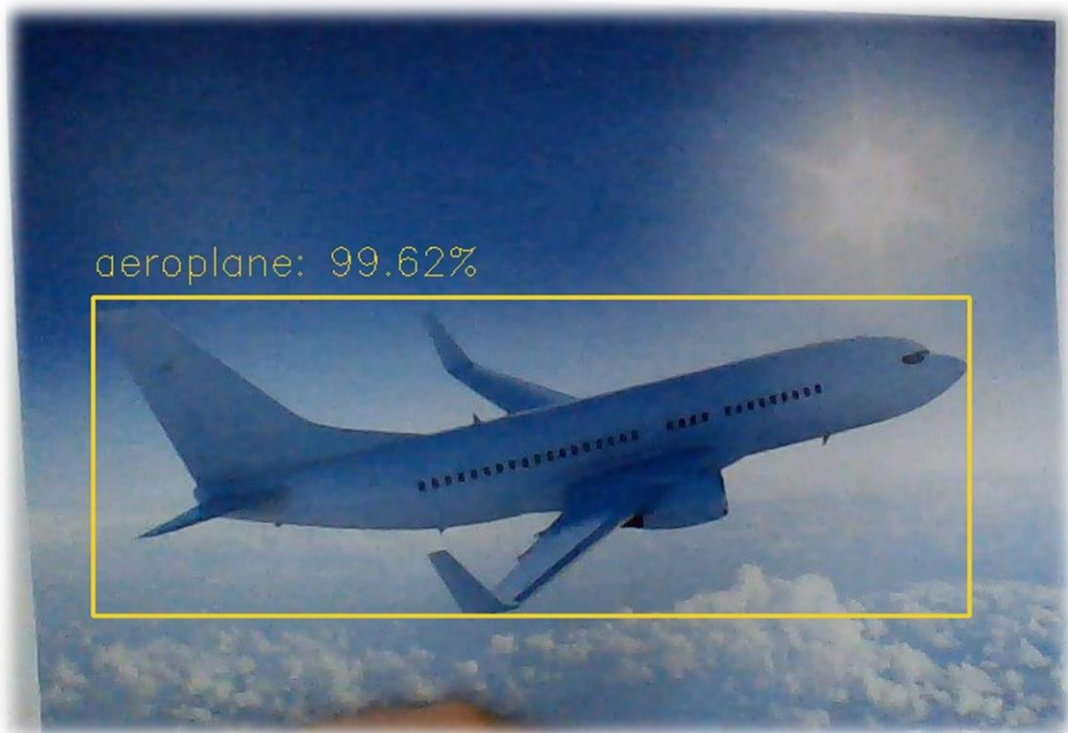


Figure 5.1.1 (d) – Successful *airplane* recognition on Nao



Figure 5.1.1 (e) – Successful *cat* recognition on Nao

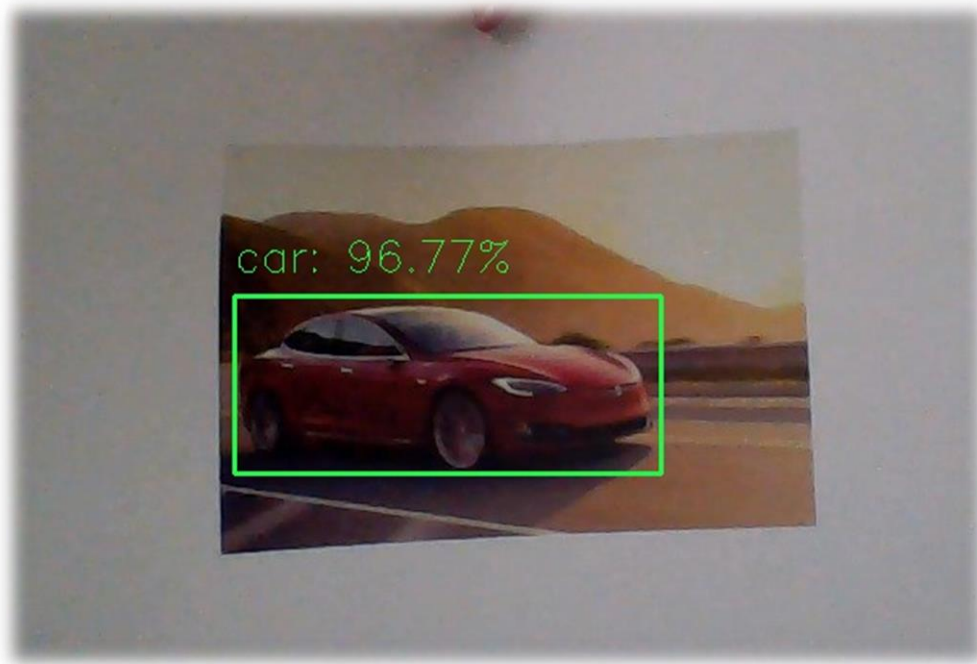


Figure 5.1.1 (e) – Successful *car* recognition on Nao

In Figures 5.1.1 a to e, there were examples of the algorithm applied on frames taken in advance on Nao, whereas the next ones represent examples of the final program, with real time detection.

Below, on the right side there is Nao's *video monitor panel*. This panel displays in real time what is seen by the active camera of the robot's head; on the left, in the console, the detected objects at that moment are listed.

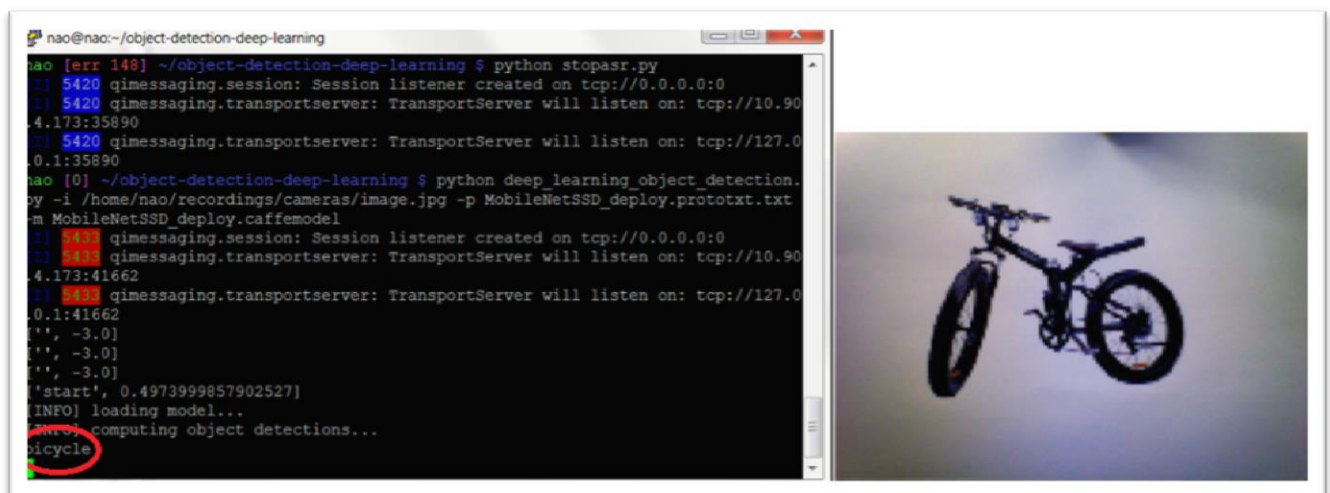


Figure 5.1.2 (a) - Successful *bike* recognition real time dection, on Nao

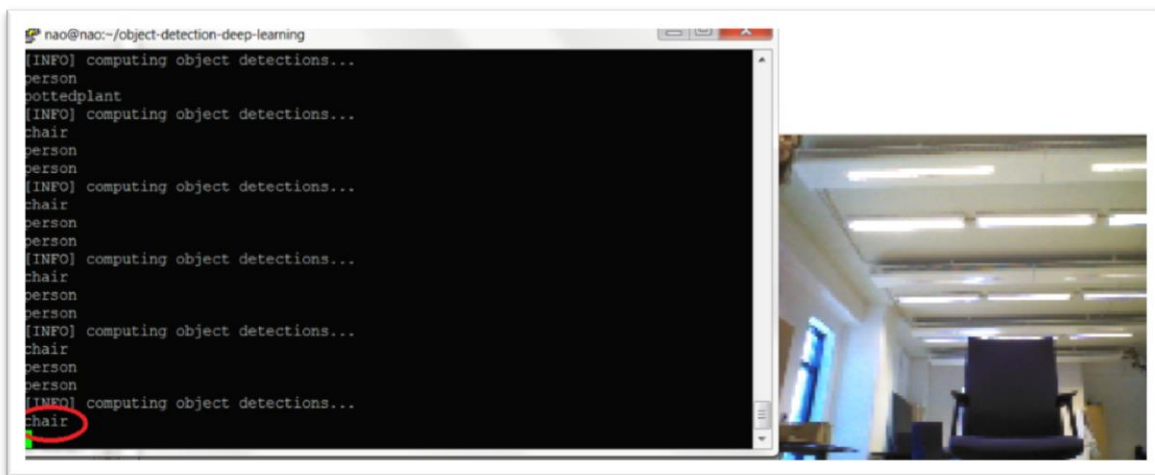


Figure 5.1.2 (b) - Successful *chair* recognition real time dection, on Nao

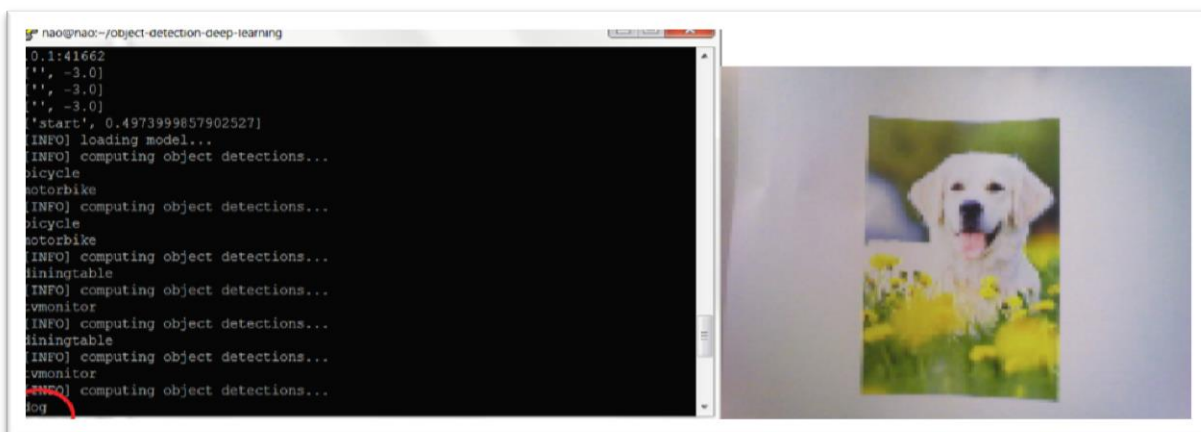


Figure 5.1.2 (c) - Succesful *dog* recognition real time dection, on Nao

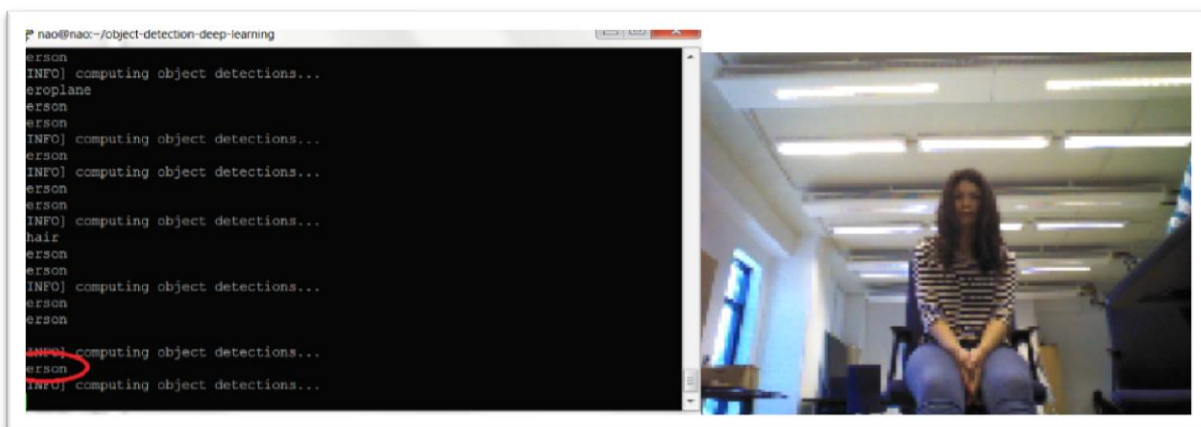


Figure 5.1.2 (d) - Succesful *person* recognition real time dection, on Nao

5.2. The performance of the Convolutional Network

For this part, several instances from 3 different classes were analyzed, in order to examine the accuracy of the implemented network on the robot. The expected result was a mean average precision of 72.7%, as the model is supposed to perform [15], but some tests with personal archive pictures was implemented to demonstrate its effectiveness.

The results are presented in Table 5.2.1 and in Figure 5.2.1 an example of the test pictures for the *cat* class is shown (27 pictures have been used for testing this class).



Table 5.2.1 - Example of test pictures

objects to be recognized	true positive	true negative	false positive	false negative	accuracy
cat	22	0	0	5	0.81
person	54	0	1	30	0.64
car	39	0	0	11	0.78

Table 5.2.1 - Results from the Caffe model

The accuracy was computed using the formula:

$$accuracy = \frac{(tp + tn)}{(tp + fp + tn + fn)} \quad (13)$$

where:

- ❖ **tp** represents the True Positive cases, namely the ones when the real cases were positive and the predicted ones were also positive;
- ❖ **tn** stands for True Negative and this number counts the cases when the real case was negative and the predicted was also negative;
- ❖ **fp** indicates the False Positive cases, specifically those when the real case was negative and the prediction positive;
- ❖ **fn** represents the False Negative cases, when the real case was positive, but the prediction was negative.

The precision can be calculated using the same notations in the following way:

$$precision = \frac{tp}{(tp + fp)} \quad (14)$$

Consequently, for the analyzed classes presented in Table 5.2.1, and taking formula 13 into account, the mean accuracy is 0.74. Similarly, using Table 5.2.1 and formula 14, for some classes, the precision is 1.

As illustrated in Figure 5.2.2, it is obvious that when the objects that are desired to be identified have a more natural position, the recognition will be successful, in contrast to the cases when only parts of them are observable.



Table 5.2.2 – Successful vs. unsuccessful *people* recognition

As the figure above suggests, if parts of objects are hidden or if they have different positions than normal conditions - meaning the training conditions – the recognition will not be successful.

6

CONCLUSIONS AND FUTURE STEPS

6.1 General Conclusions

The main objective of the thesis was to develop a program to help the humanoid robot Nao recognize objects around and take actions autonomously, depending on what is detected; this goal was accomplished and some improvements were made.

The final application comprised of three parts: training the network, testing it and detection plus recognition. In this way, the implemented program enables Nao to recognize up to 90 classes; thereby, using the other pre-installed features on the robot, it will be able to take actions autonomously, depending on what it sees.

In its current state, Nao is able to perform object detection and recognition, using vocal commands to start and stop the program. For this project, convolutional neural network were used, specifically a combination of MobileNets and SSD.

The implemented algorithm works in the following way: after launching, Nao waits for a start command; when given, it begins the main program by taking frames of the surrounding. From the frames that the robot takes, the algorithm reports the presence of objects around it, using one shot at a time. The used architecture, MobileNets, especially made for embedded applications, assures a compromise between latency and accuracy, while the SSD MultiBox technique ensures a real-time detection. Using this approach, the algorithm performs a real-time classification and outputs the identified object, therefore making it capable of decision making, depending on the output.

This translation from the three-dimensional world into a particular behavior increases the freedom of the robot, making it able for autonomous human interactions.

6.2 Personal contributions

In order to achieve the proposed goal, my personal contributions for the final algorithm were the following:

- ❖ importing the necessary libraries of OpenCV3 on Nao
- ❖ importing the pretrained model with 20 classes on the robot
- ❖ training the network for 90 classes and importing the new model on Nao
- ❖ implementing the speech recognition and the text to speech modules in Python, starting from the modules offered by Aldebaran
- ❖ integrating the speech recognition, the object recognition and the text to speech modules into the final program, in Python

6.3 Future work

With the high evolution nowadays in the robotics and autonomous systems area, the project presented in this thesis is just a first step towards making Nao less dependent on humans. Since research is actively carried out in this domain, it is just a matter of time until free source portable recognition programs will be available to general public.

An interesting step towards innovation would be training the network that is on the robot for other objects recognition; these objects could be specific for certain applications, thus enabling users to have a robot able to detect for instance when certain people enter a room. A similar scenario would be training Nao to recognize people on the ground, thus making it very useful as elderly assistant. Apart from emergency cases, the robot could be a real help for old or even blind people, by recognizing traffic signs and lights or specific features of the road.

Another good implementation of this thesis' project could be helping children with mental disabilities in therapy modules. It is a well-known fact that working with children with mental illnesses is not an easy task and recent research showed that introducing small robots like Nao in some therapies might have a good impact on the overall results.

Looking towards the future, Nao could be a great help either as an assistant, teacher, or simply as a tool for making children interested in technology; and with the great advances of Artificial Intelligence, it could become a powerful device in every home.

References:

- [1] [Nao's Modules] Aldebaran Robotics website (<http://doc.aldebaran.com/1-14/dev/>, accessed on 20th October 2017)
- [2] [Goodfellow, 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, 2016, Deep Learning, MIT Pres
- [3] [Stanford Course, 2017] Fei-Fei Li, Justin Johnson, Serena Yeung; 2017, Course Notes, CS231n: Convolutional Neural Networks for Visual Recognition , Stanford University (<http://cs231n.github.io>, accessed on 24th May 2018)
- [4] [Adam Harley's application] On-line application for a convolutional network layers' visualization (<http://scs.ryerson.ca/~aharley/vis/conv/flat.html> accessed on 1st June 2018)
- [5] [SE Network paper, 2018] Jie Hu and Li Shen and Gang Sun, 2018, Squeeze-and-Excitation Networks, IEEE Conference on Computer Vision and Pattern Recognition
- [6] – [MobileNets paper, 2017] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H., 2017, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Application
- [7] [Batch Normalization paper, 2015] Ioffe, Sergey & Szegedy, Christian, 2015, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- [8] [SSD paper, 2016] – Liu W. et al. SSD: Single Shot MultiBox Detector, 2016, In: Leibe B., Matas J., Sebe N., Welling M. (eds) Computer Vision – ECCV 2016. Lecture Notes in Computer Science, vol. 9905, Springer
- [9] Matt Harvey, Jul 11, 2017, Creating insanely fast image classifiers with MobileNet in TensorFlow
- [10] – [OpenCV Shape Detection], Adrian Rosebrock, 2016, Image Processing, OpenCV 3, Tutorials (<https://www.pyimagesearch.com/2016/02/08/opencv-shape-detection>, accessed on 20th November 2017)
- [11] [Google Open Source Blog], Jonathan Huang, 2017, Google Open Source Blog (<https://opensource.googleblog.com/2017/06/supercharge-your-computer-vision-models.html>, accessed on 20th January 2018)
- [12] [COCO dataset] (<http://cocodataset.org/#explore>, accessed on 3rd December 2017)
- [13] [Tensorflow detection model zoo]

(https://github.com/tensorflow/models/blob/477ed41e7e4e8a8443bc633846eb01e2182dc68a/object_detection/g3doc/detection_model_zoo.md, accessed on 25th November 2018)

[14] [Caffe deep learning framework] (<http://caffe.berkeleyvision.org/>, accessed on 3rd December 2017)

[15] [MobileNet] (An implementation of the MobileNet architecture, available at <https://github.com/Zehaos/MobileNet>, accessed on 12th December 2017)

[16] [MNIST dataset] (available at <http://yann.lecun.com/exdb/mnist/>, accessed on 6th December)

Annex 1

