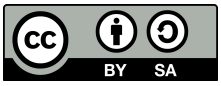
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS DE TELECOMUNICACIONES

TRABAJO FIN DE GRADO

Deep Learning Applications for Robotics on TensorFlow

Autor: Ignacio Condés Menchén
Tutor: Dr. José María Cañas Plaza
Curso académico 2017/2018



Summary

Nowadays, continuous improvements on Computer Science allow to address more complex tasks than traditionally. So, we can begin to artificially handle more *human* tasks, which are performed on a more efficient way when the processing structure is modeled *emulating human brain*. This field of study is covered by *deep learning*, which particularly in the Computer Vision field makes the difference between a exhausting analysis of *designed* features (which can be susceptible to environmental transformations or distortions), and *automatically* extract abstract features, allowing a *robust* operation, which can be executed on a *real time* manner.

On the other hand *robotics*, gradually more present in daily life, is more accessible, compatible and interoperable. This leads into a faster deployment of robots: not so long ago we only had huge, complex and not practical robots on production lines. Now we have autonomous vacuum cleaners perfectly able to clean our house and go back to its dock, at perfectly affordable prices for the vast majority of people.

There is a really interesting *synergy* between these two fields, which allow to combine the *perception* skills that a *deep learning* system can achieve, with the wide variety of physical *responses* that a robot can perform.

In this work, focused on introducing new *deep learning* Computer Vision applications in the academics and research platform JdeRobot, three real-time components have been developed, making use of *neural networks*: a *digit classifier*, a generic *object detector*, and a *person following component*, which also incorporates a *reactive behavioral to follow a specific person*. This is achieved combining an RGBD sensor, detection and facial analysis respective neural networks, and a robot equipped with wheels.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introductory chapter aims to present the general context which wraps this project, going in some depth inside *deep learning and robotics*. We will also amble along some of the latest advances and most useful current applications of the junction between these two fields. Lastly, we will situate this project on the previously described context. After this chapter, each key aspect of the work flow will be further explained.

1.1 Robots

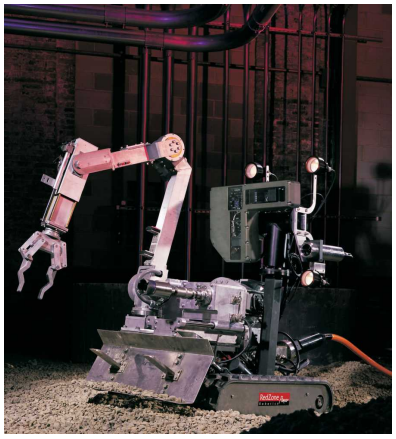
Robotics applications can be really useful at daily tasks. These tasks are of greater interest when the behavioral of a robot tends to emulate the human one[1], with the advantage of no human beings exposed to a significant risk, or, in a less gloomy scenario, without human body physical limitations. This requires a polished (and somehow complex) behavioral, which is triggered by a certain input. At this point, we can find two main branches into robots, depending on the input source:

- **Teleoperated robots:** this kind of robots are capable of perform certain actions, which are *remotely controlled by a human operator*. This application is the one with most weight on the hazardousness (Figure 1.1a) [6] or precision [7] factor. Thus, some advances are made nowadays improving the teleoperation function, implementing *feedbacks* from the robot, such as haptic feedback [8], or VR (*Virtual Reality*) sensation, to allow that person to sense the environment as if it was in front of her.

- **Autonomous robots:** these robots are much more complex machines, as they are distinguished for implementing a response by itself, independently of any kind of remote operator. This is seeked on certain scenarios, where the time elapsed performing an action or the cost of mantaining a critical link with a base, are factors with a considerable weight in the design [9]. This is the kind of robots that concern us on this work: the state-of-the-art techniques try to emulate *human behavioral*(Figure 1.1b), so some actions can begin to be performed with a certain intelligence, as we will describe below.

The important advances on the last decades on the image processing and audio recognition fields have impulsed the development of assistance systems, apart from critical machines as

[1] Some efforts are taken even into adopting the performance of human's best friend



(a) Pioneer robot, designed to perform hazardous teleoperated explorations in a deadly radioactive environment.

(b) Pepper, an autonomous humanoid capable of performing on board processing and reaction to external stimuli on a human way.

Figure 1.1: Robots of each described kind.

the previously described examples.

This way, several applications have arisen on people recognition and conversational behaviors, and it has been spread to everyday purposes, from personal assistants[2], to autonomous driving[3].

1.2 Deep Learning

1.2.1 Machine Learning on Computer Vision

Almost every time, the desired behavioral is one or more deliberated or reactive responses[4], triggered by a certain input (typically perceived by on-board sensors, among others). This raw data, which is typically retrieved on a simple way (images, audio), is processed and mapped into a concrete response. At this point, we can bring up the key question: *how do we process the raw input to obtain a suitable action for the current requirements, or needings?* The answer for that question is *machine learning*: the computer science field that pursues the capacity of machines to learn the suitable response to a previously unknown input. This is achieved by

[2] Google Smart Home

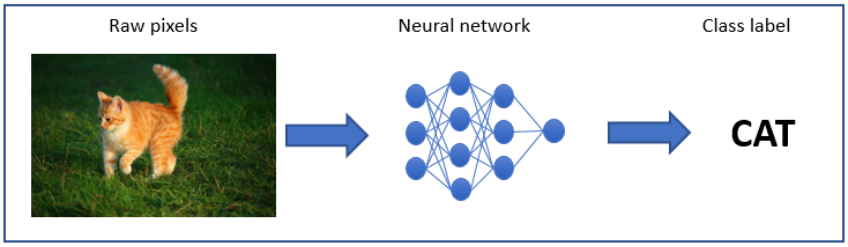
[3] Tesla Autopilot

[4] A *deliberated* response implies a certain level of *extra intelligence*. It figures out which could be the best action to perform, considering present, past and probably future information to make the decision.
On the other hand, a *reactive* response makes an immediate decision, depending just on what has been just perceived.

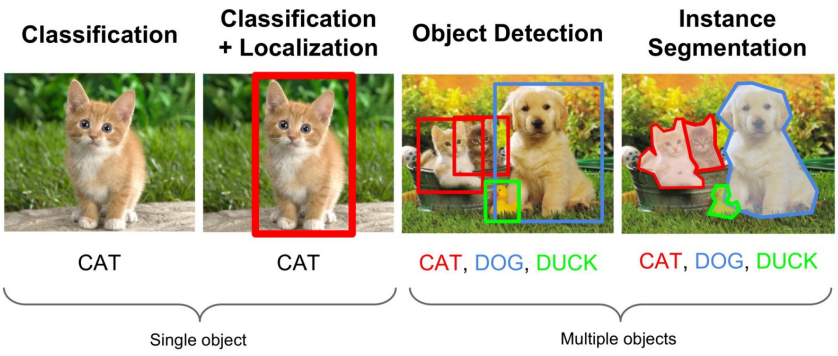
performing a training with a dataset of examples, which need to be properly formatted: the system has to previously know what to look for and evaluate, what is typically called *features*, and learn the proper parameters for an optimum output.

Generally, machine learning applications on image processing can be split into two types of response (Figure 1.2):

- **Classification:** given a set of possible classes $\{c_1, c_2, ..., c_n\}$ to which an image x_i can belong, we select the class c_i where x_i fits the best, given a set of features extracted from it.
- **Detection**: given an image x_i, we decide if we can find or not an object/region inside of it which fits into the searched type. In the affirmative case, we locate it (using a region or a bounding box).



(a) Classification



(b) Detection

Figure 1.2: Functional difference between *classification* and *detection*.

1.2.2 Neural Networks

This has turned deep learning into the cornerstone of current *AI* applications, which don't need complex dataset with a lot of preprocessing (that require important human effort) anymore. That simplicity is achieved through the use of *Neural Networks*. A Neural Network is the representation of an algebraic algorithm which implements non-linear calculus models [10]. It is composed by several processing *layers*, which are made up of *perceptrons*, that are generally called *neurons*. This is because these neural structures *emulate the human brain*, formed by a huge set of interconnected neurons, which are disposed on the already mentioned

layers (Figure 1.3).

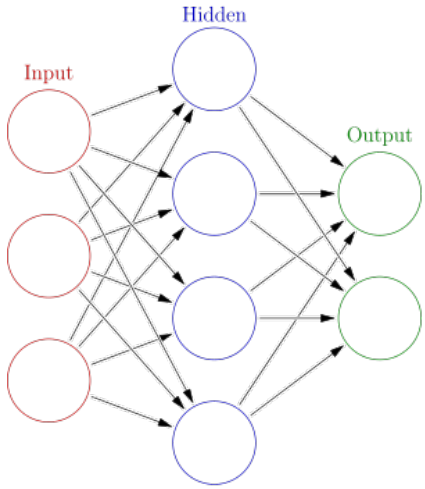


Figure 1.3: Structure of a Neural Network

First approaches to neural networks, according to [11] were developed on the 50s-60s decades. This was when the computational potential allowed to develop on a real machine the first modeling of the way it was believed that a brain neuron works, which was inspired by electrical circuits. These experiments [12] were performed by the neurophysiologist W. McCulloch and the mathematician W. Pitts. Later, in 1949, Donald Hebb [13] observed that the synaptic path between two neurons is reinforced (its efficiency rises up) every time it is used. This introduced the concept of *training* on a neural network.

1.2.3 Processing unit: the perceptron (neuron)

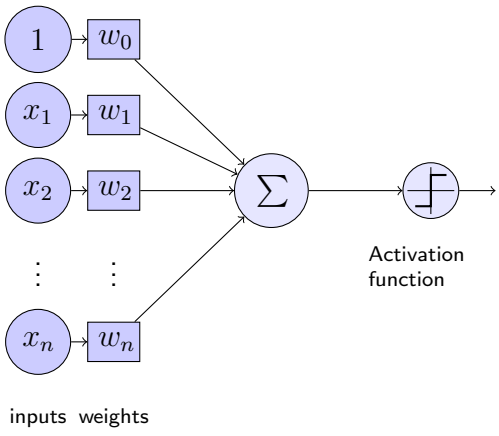


Figure 1.4: Diagram of a perceptron/neuron

Every neuron is composed by an structured schema:

1. *Inputs*: the data which come into the neuron. It might come from the main stimuli, or from another neuron (as the output of the previous layer)

2. *Weights*: the tuned parameters of the network. They represent the importance given to each feature on that singular neural unit. The weight w_n multiplied x_n times results on the contribution of the feature n in the current neuron.

3. *Sum*: the product of all the inputs with their suitable weight come into a sum operation[5], to *build a total linear response:* $z = \sum_{i=0}^{n} x_i \cdot w_i$[6].

4. *Activation function*: this is an important part of a neural network as, until this moment, all the numerical computations we have performed were just linear operations. If we keep the output of the neuron being a linear function of the input, we will lose the effect of having more than 1 layer, as really the total result of all the network is a linear function of the first input, so we could simplify all the network down to one single neuron.

   For this reason, we use a *non-linear activation function*, which maps the linear combination computed by the sum, into the $[0, 1]$ interval, on a non-linear function. A typical function is the one called *ReLU* (REctified Linear Unit) [14], which follows the formula:

$$g(z) = max(0, z) \tag{1.1}$$

5. *Output*: when the activation function has been computed, it is forward-propagated to the output, or to the neurons belonging to the next layer. As it has been said, it is mapped into the $[0, 1]$ interval, so it can be seen as the importance that particular feature will have on the next neuron: if it takes nearly zero values, the next neuron will be poorly stimulated. There lies the meaning of the name of the previous component: *activation function*.

1.2.4 Deep Neural Networks

Deep learning is the piece of machine learning that is capable to *automatically learn the features that the system could use from primary data* (pixels on images, samples on audio, words in text processing, etc.).

The fact of having more than one layer gives to the network the concept of *depth*. This opens the door to a vast set of possibilities, as *it allows us to perform deep learning with neural networks: Deep Neural Networks*. This can be achieved, as we can see on Figure 1.5, by introducing a new kind of layer, where all the new neurons are connected to every single neuron of the previous one.
This is typically called a *fully connected layer*, and the fact of relating every single activation from the previous layer with a set of tunable weights on each neuron allows to rapidly find common patterns followed by features seen on one of the analyzed scenarios (e.g. syntactical relationships between several kinds of words in language processing, or finding edges or shapes on image detection/classification).

[5]We consider w_0 as the product to the constant input 1, as the intercept term (a constant always present independently of the current input).

[6]There is also a summation bias term on each neuron, b_i, but it is ignored here for the sake of simplicity, as the weights are more representative with respect to the input.

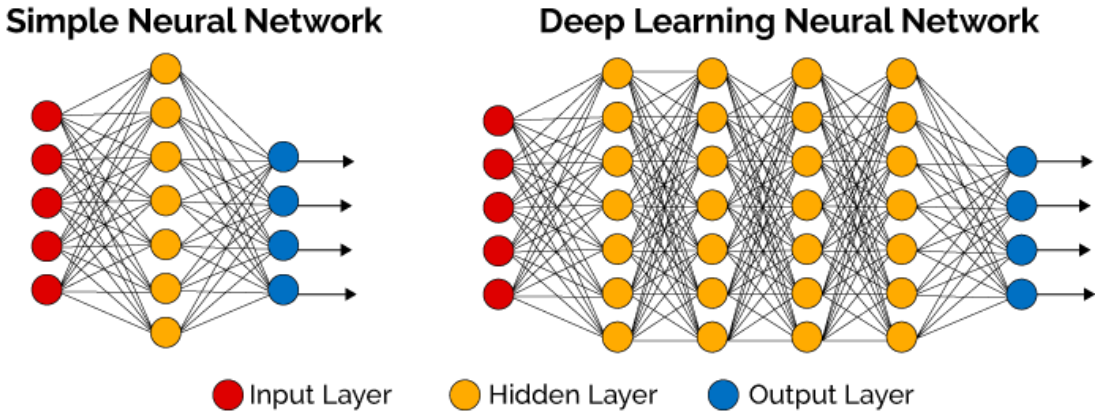


Figure 1.5: Evolution to a *deep* neural network.

1.2.5 Convolutional Neural Networks (CNNs)

Finally, this leads us to the last concept we will study on this dissertation. As we have said before, we can connect a big set of neurons between themselves to extract more abstract and complex features, of increasing interest with the number of neurons and layers.

If we aim to apply this processing to images (*Computer Vision*), we have to take into account that, if we want to input an image into a neural network, each pixel has to be taken as an input, and also the fact that an RGB image is composed of 3 channels (1 channel per color), so, for an image with a dimensions of m pixels wide and n pixels high, we will need $m \cdot n \cdot 3$ input neurons. Besides this considerable number, we will have to take into account the neurons resulting on the additional deeper layers that we will add to have an high enough abstraction level for our application. This drives to absurd numbers of simultaneous neurons working, that are difficultly handable during a feed-forward execution, but absolutely unfeasible on a training process. An additional problem can be a moving object/region on the image: we must be capable to detect the shape of a car on the right side of the image, or in the left one.

We can solve both problems simultaneously with an easy procedure: we will not process the entire image at one time. Instead of that *we will perform a convolution operation (Figure 1.6) between the inputs of our network, and different regions of the image*, sliding and multiplying a *parameterized mask* along the whole image. This operation is performed with the objective of the product returning a high value on the interesting regions of the image. This will output *activation maps*, which symbolize the response of that portion of the image to the weight mask. This can be performed, as we have said before, a few times with different masks to obtain features with a higher degree of abstraction. As we want to keep the computational complexity low, we can alternate these layers with *pooling layers*, which subsample the resulting maps, to keep it simple (if we keep only 1 of each 3 pixels of an activation map, selecting it carefully to retain the maximum information, we can reduce the number of necessary neurons on the next layer on a factor of $\frac{1}{3} \cdot \frac{1}{3} = \frac{1}{9}$). This is reflected on Figure 1.8, where the process of convolution-pooling can be repeated a few times, and then the result (which should not have considerably big dimensions) are inputted into a fully connected layers, to extract and handle the relations between the features and the possible classes (on a classification scenario).

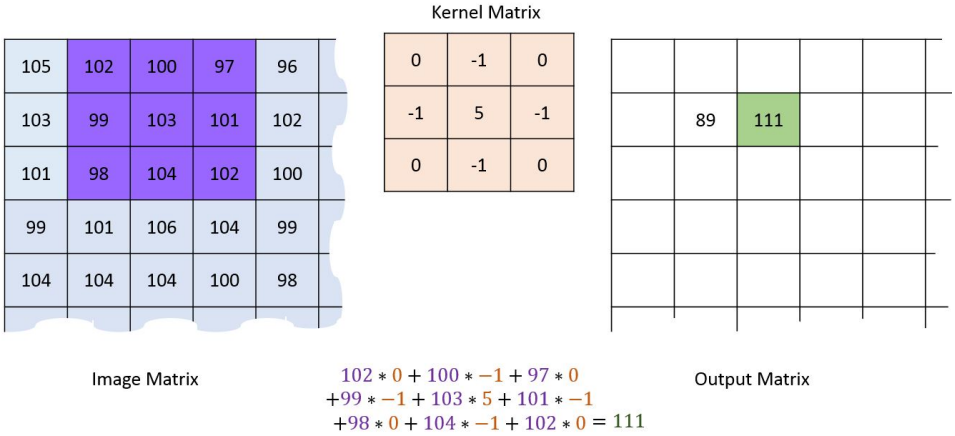


Figure 1.6: Convolution operation applied on an image (image from [1]).

But, *how do we find the best value for each weight, and for each layer?* That's the process we call *training a network*. Using a technique called *back propagation*, we can compute

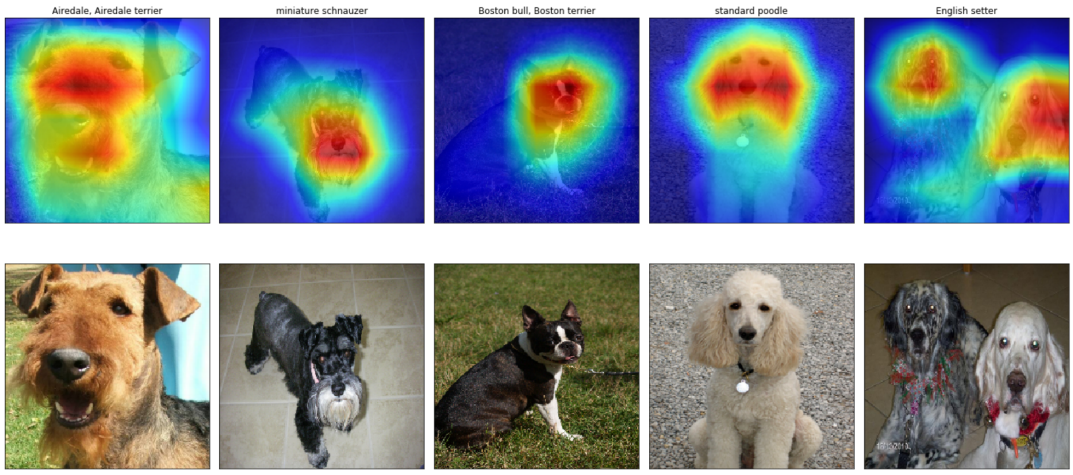


Figure 1.7: Activation maps of a detection CNN searching for dogs on different images [2].

1.3 Deep Learning on JdeRobot

So, as we have been describing, Deep Learning can be of a great interest on the image processing field, as it allows to implement an easy and really robust AI algorithm.

JdeRobot[7] is an open-source software development suite, built from this University, and among all the developed software/investigation inside it, we can find some interesting programs/projects for our purposes:

- `Detection Suite`[8]: it is a C++ application, suitable to load/benchmark *detection*

[7] https://jderobot.org
[8] https://github.com/JdeRobot/dl-DetectionSuite

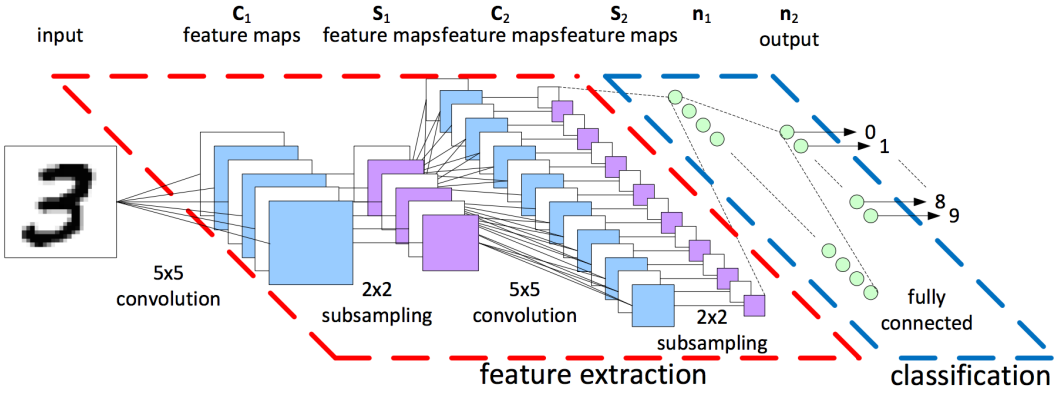


Figure 1.8: Schematic of a CNN.

Darknet/YOLO[9] models, against different databases. It is also capable, through a Python→C++ interface, to load TensorFlow/Keras models as well.

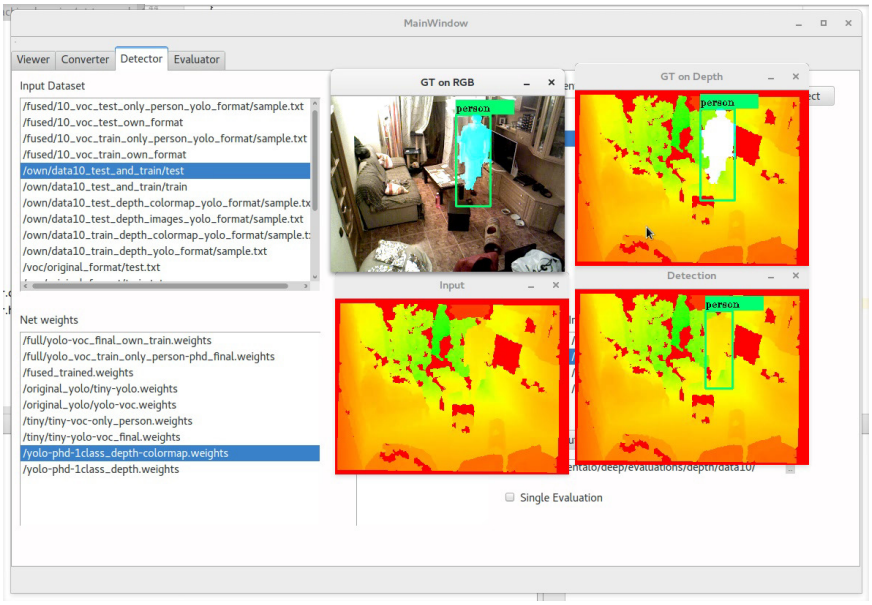
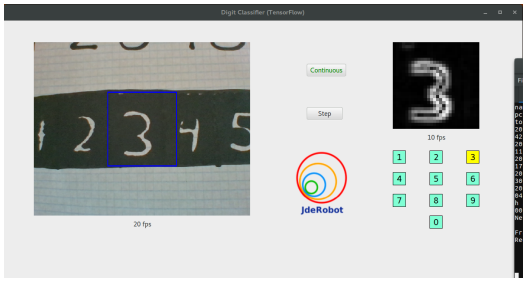


Figure 1.9: `DetectionSuite` on action.

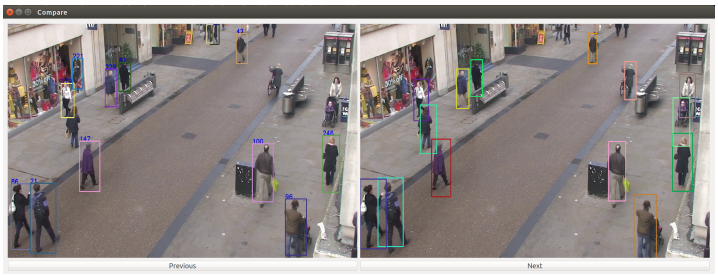
- Final project of David Pascual [15] and Nuria Oyaga [16]: a further study of Deep Learning, applied on Python (Keras and Caffe frameworks, respectively) to *digit classification* (Figure 1.10a) implementing a CNN as it has been seen.

- MsC project of Marcos Pieras [17]: On this master thesis, an application implementing two neural networks (as we will do) has been developed. One of them allows us to *detect* people on an image, and the other one (a siamese network, as we will describe later) can track features of each person, to keep every detected individual identified on a surveillance image system (and possibly trace the route followed by each person) (Figure 1.10b).

[9] https://pjreddie.com/darknet/



(a) `DigitClassifier` working.



(b) `PeopleTracker` working.

Figure 1.10: Some JdeRobot student projects on action.

In conclusion, as we have been mentioning and been taking a glance on the possible applications, Deep Learning can make such a brilliant tandem along with a reactive behavioral. We have taken a glance on a few possible applications, and the following dissertation will struggle to demonstrate it.

Robotics + deep learning rock!

Chapter 2

Objectives

Once we have presented the introductory context of this project, we will describe its objectives, along to the followed methodology to achieve all of them.

The final objective is to enrich the JdeRobot platform on its Computer Vision aspect, upgrading an existing component and creating two consecutive new ones. Keeping this in mind, we will be able to generate a behavioral focused, as a specific application, on tracking and actively following a person, making use of a robot. This internal process of transformation of a stimulus into a reactive movement will be accomplished using *Convolutional Neural Networks*.

2.1 Main objectives

2.1.1 Classification

Our first objective (and the first task to tackle) will be to upgrade the support of the digit classification tool already existent in JdeRobot, `DigitClassifier` (subsection 3.4.1). This will allow us to augment the scope of this component, due to the support for the new framework, TensorFlow (section 3.7). This is a good starting point to achieve some initial skills building and training Convolutional Neural Networks on TensorFlow (it will be the main framework used all along the project).

2.1.2 Detection

As it will be described on the suitable chapter, we will build a component (`ObjectDetector`) which deploys *a generic object detection algorithm* on an incoming video stream. This component will be ready to work in *real time*. It will also be compatible with new network models (e.g. a new detection model trained by us), which will be loaded transparently at runtime.

As it can be inferred, it will not provide a response *per se*. Its visible output will be to draw *bounding boxes* surrounding each detected object, indicating as well the class where that particular object belongs (person, airplane, dog, etc.), and its score (confidence in %).

2.1.3 Tracking and following

As an example of the plethora of possible applications of the previous described objective/milestone (object detection on an image), we want to implement a "person following" behavioral. Our main objective here is to *identify and track* the person to follow, which will semantically be called *mom*. The component that comprehends the previous detection behavioral and this new one will be named `FollowPerson`. The great advantage here is the strength a CNN can achieve under variable light conditions. That makes this technique perfectly suitable to command physical actuators on a robot.

We will make use of the detected people (with the technique followed on the previously described node and constraining the result to only retain people detection), and look for the face of each one of them. Later, we will make use of another neural network technique, called a *siamese network* (it will be properly explained later) to identify them. It will allow us to find *mom*, in case it is being seen by the camera, and command a proper response to the robot with the objective of following *mom*.

As we have said before, these two last nodes/components are successive. Thus, they will share an important part of the global objectives.

We will start breaking down the common objectives between both applications, into three functional blocks:

- *Design:* both milestones have been preceded by a first *documentation* phase. While the theoretical base was learned and achieved, some papers and examples were investigated. We will go some deeper in each section.

  Later, a more specific design phase will be to specify the structure of the nodes. From now on, we will have to perform several tasks simultaneously:

  - Grabbing the incoming image(s) from the sensor(s).
  - Processing the image(s).
  - Properly update the GUI (*Graphical User Interface*) with the image(s) and the outputs of the processing.
  - *Only on the following node:* send the computed response order to the actuators.

  These tasks should follow an *asynchronous schema* every time, to avoid blocks between tasks, and taking advantage of some shared memory to exchange inputs/outputs. Besides, this allows us to have a custom iteration period for each task: we shouldn't have to wait until the neural network finishes feed-forwarding the image to refresh the *GUI*. On a high level structure, it could follow the schema on the Figure 2.1.

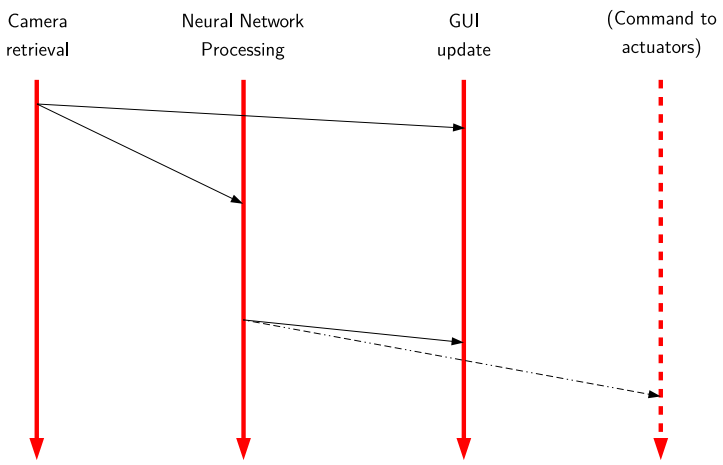


Figure 2.1: Parallel tasks to perform, and data exchange between them.

- *Implementation:* the next and most time consuming objective is to *develop both components*. The advantage is, once again, that both components are successive. This makes code reutilization a very interesting move, as we will only need to perform a language-friendly additions (on a simple way, thanks to *Object Oriented Programming*)) to the `Object Detector` code to implement the `Follow Person` features (depth images support, computing and commanding movements to the motors, among others). Details will be described on the appropriate section.

- *Experimentation*[1]*:* these nodes have a very strong tunability component (from neural network parameters to movement factors in the commands for the motors, stopping over describing all the desired behavior that the following algorithm has to adopt in several situations that it has to be capable of handling).

Finally, we will briefly highlight the *personal* objectives we have pursued on this project. As this has been developed along a whole year, and on a relatively abstract field as *deep learning* is, a level of rigour has been necessary to accomplish satisfactory results. This provides a novice investigator the opportunity to learn about the phases and development process on a much more professional way than a homework task.

In addition, this project has allowed an interested person in *deep learning* to learn about a cornucopia of concepts and experience. Later, when it was decided to evolve towards creating a reactive behavioral, it was motivating to make the most of a possible synergy between two different fields of knowledge, as *deep learning* and *robotics* are. It has to be remarked that the implementation of the development has always been possible, so it has been easy where there was more work to do in every moment.

[1] *"In theory, theory and practice are the same. In practice, they are not."*, Y. Berra

2.2 Methodology

The development of this project, as it has been described, has been subdivided into smaller tasks, or *prototypes*, which could be addressed as individual tasks to achieve. The way to tackle them has been a *spiral methodology* [18].

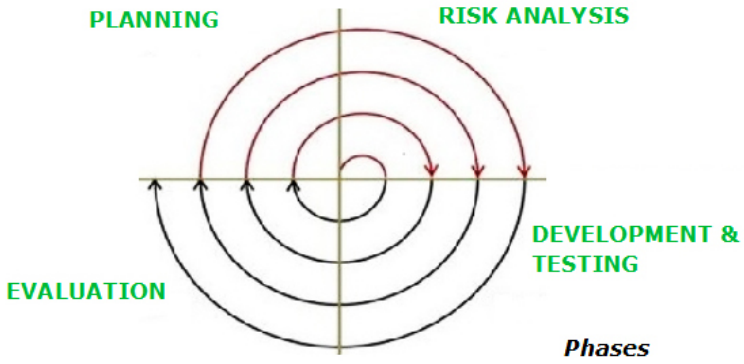


Figure 2.2: Spiral Development Model.

This consists on a software development work procedure that, on a general outline, is very similar to a conch. It describes a 4 phases methodology [19], explained right below:

1. *Planning:* establishing the objectives to tackle on the incoming work iteration.
2. *Risk Analysis:* Later, we evaluate the possible risks and dangers we can find developing the specific program(s). For each found risk, we will try to find a solution to solve or, at least, mitigate it beforehand.
3. *Development & testing:* This phase is purely focused on writing the planned piece of software, following the guidelines obtained on the previous steps. In addition, corresponding tests should be performed to check that the work will accomplish the asked functionality.
4. *Evaluation:* Lastly, when the development phase has been finished, an evaluation has to be performed on the results. This will be the key to know if it is compliant with the initial requirements and if, hence, its development has been successful.

As this was the general procedure followed for each iteration of the developed software, the completion of the evaluation phase immediately led to another planning phase, already belonging to the next iteration. As this is a cyclic process, we can perform as many iterations as desired, slightly increasing the scope of the project on each new one.

The workflow present on this project has been supported by weekly meetings, scheduled in order to get up-to-date with the last established objectives and tasks, and set up the work until the next one. This has allowed to keep a constant feedback with the tutor and hold the followed path onto the desired direction.

Additionally, a MediaWiki page[2] has been mantained on the JdeRobot website, reflecting every effort and achievement in order to have a good temporal reference of the work done, and a timeline of accomplishments, and including demonstration videos for each successful iteration result.

The code for all the project has been handled on the GitHub repository[3] created for this purpose. However, as the resulting nodes were officially incorporated to the JdeRobot environment, they were migrated to their own repositories. This will be further described on the section dedicated to each component/iteration.

2.3 Requirements

For every covered topic, the developed solution must be compliant with the requirements formulated below:

-

[2] https://jderobot.org/Naxvm-tfg
[3] https://github.com/RoboticsURJC-Students/2017-tfg-nacho_condes

Chapter 3

Infrastructure

3.1 Hardware

3.1.1 Sony EVI D100P

(a) Front side.

(b) Back side.

Figure 3.1: Sony EVI D100P.

The first hardware element that we use is the Sony EVI D100P[1]. It is a *PTZ* cam (which stands for *Pan Tilt Zoom*). It is a camera which, originally thought and designed for video-conferences, is equipped with a bunch of precision motors. This allows it to be teleoperated, performing a soft and steady two-dimensional movement on demand:

- *Pan:* horizontal movement. It can take values from $-164°$ to $164°$ from the centered position. This movement can be performed at a certain speed, which can be setted between 1 and 24.
- *Tilt:* vertical movement. Its range goes from $-30°$ to $30°$, and the movement speed can be also varied between 1 and 20.

The low-level implementation of the movement commands is the *VISCA* protocol, a proprietary solution from the manufacturer (Sony). It is received by the cam through a RS-232C (the traditional low-rate serial interface before USB extended), so we can connect it to a modern computer with a RS232-USB interface.

[1] https://pro.sony/en_IN/products/ptz-network-cameras/evi-d100-d100p-pal-

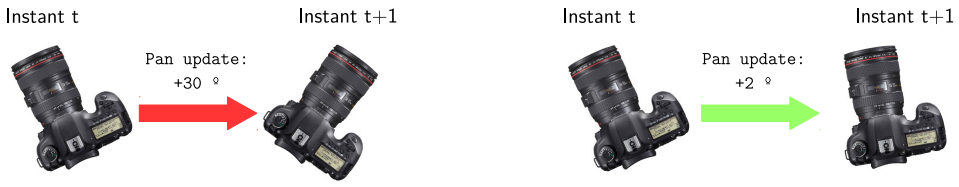
However, the driver that controls this camera (3.4.2) does not offer support for a *zoom* movement, but it is not very relevant for this application.

As the video sensor is an analogue device, we need to convert it to a digital format. We achieve this with a video capture device, which outputs video in a digital format. This image flow is processed by a ROS driver (subsection 3.3.1), that will be later explained.

Something remarkable about this device is that it is a bidirectional device: we *receive* images from its camera, and, at the same time, we *send* it commands to move the motors.

As we can infer from the described technical specifications, this is a relatively old device, so we have to be careful on the movement commands. This is due to the short period of position update: even if we command the motion action at the maximum available speed, the movement won't complete before the next update is commanded. In addition, the camera maintains a buffer of the pending commands to be updated. Hence, sending absolute movement commands (Figure 3.2a) will result in a chaotic behavior of the camera.

So, we need an alternative approach, consisting of *differential* movements (Figure 3.2b). This has the objective of ensuring its completion before the next iteration comes in, so we perform it at the maximum available speed.



(a) Wrong motion update (too long movements).

(b) Correct motion update (short differential movements).

Figure 3.2: Comparison between possible approaches for Pan/Tilt angle updates.

This is the device we use on our first approximation to the sensing+actuating node (section 6.7), where the only response is moving the camera.

3.1.2 Asus Xtion Pro Live

It is a RGBD (RGB + Depth) sensor, designed by Asus for interactive PC applications development purposes. We use it as the imaging source in the developed sensing+actuating node (chapter 6).

Figure 3.3: Asus Xtion Pro Live. IR emitter (left), and RGB and IR lenses (right).

It counts on the left side with an IR (*infrared*) light emitter, which radiates beams like a conventional light bulb (that's its function).

On the right side, we can find two sensors:

- *RGB sensor:* a conventional digital camera, with a resolution up to 1280x1024 px.

- *Depth sensor:* it is capable of measure distance to objects, by receiving the reflections of the IR beams that we have mentioned above. It maps, for each pixel, the distance to that reflection (in mm), stored as a 16-bit long value.

  Thus, we can obtain a depth image, with a resolution of 640x480 px (@ 30 fps).

As it can be seen on Figure 3.3, both sensor can't physically be in the same place, so there is a little discrepancy between both computed images:

With the goal of paliating this disparity, a process called *registration* is executed for every new incoming depth image. It consists of a projection of the depth pixels into the RGB image, trying to align on an optimum way each pixel with its counterpart on the RGB image. We can observe that this can cancel to some degree the difference between both images (Figure 3.5).

If we compare the new disparity (Figure 3.5) with the previous disparity (Figure 3.4c), we can appreciate that now, the RGB and Depth images are aligned on an improved way, as if both sensors were on the same place, or much closer at least.

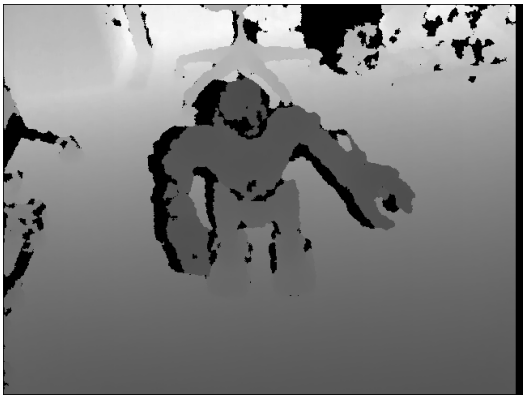
So, from now on, we will call *depth image* to the registered version of the depth map, as the unregistered image is not useful anymore.

At last, we will make a mention to the open source drivers[2], OpenNI (*Open Natural Interaction*). They were originally developed by the Kinect developer company PrimeSense (which designed the Xtion device beside Asus).

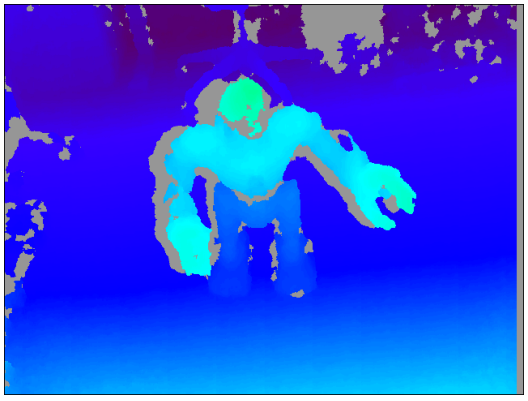
This interface allows to perform all the processes involved into handling this device (image grabbing, depth registration, etc.).

[2] https://structure.io/openni

(a) RGB image.



(b) Depth image.



(c) Disparity (difference) between 3.4a and 3.4b.

Figure 3.4: Both images sensed by the Xtion cameras, and the disparity between them *before* the registration process.

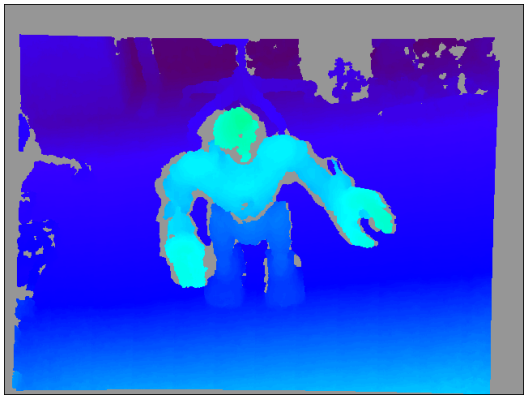


Figure 3.5: Disparity between the images *after* the registration process.

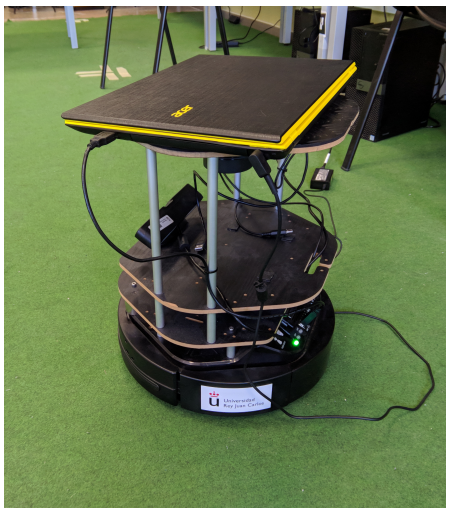
3.1.3 Turtlebot 2

The Turtlebot platform is our main actuation platform on the developed actuation+response node (chapter 6).

It is a research platform, composed by a structure jointed to a Kobuki robot (mobile base)[3].

[3] http://kobuki.yujinrobot.com/about2/

(a) Frontal view.



(b) Side view.

Figure 3.6: Turtlebot development kit.

According to its technical specifications[4], it can catch up speeds of $700\ mm/s$ (on straight line), and $180\ deg/s$ (turning).

Into the attached structure, we can find mounted an Asus Xtion sensor (in the original model of the Turtlebot 2, it is a Microsoft Kinect instead), and a laser sensor, which is not used on this project (nevertheless, it could be a continuation, as a navigation algorithm could be added to this project, on a similar way than in [20]).

The user has the capability of connecting each of these devices via USB to the laptop, and place it at the top platform of the robot. From there, the computer can run the algorithm and command the movements. Every component can be handled with the respective ROS driver (which will be described later).

3.2 Python

According to the official definition from [21], Python is *an interpreted, object-oriented, high-level programming language with dynamic semantics*. It was created in 1991 by Guido van Rossum (who consecrated its name to the TV series *Monty Python*). However, due to the increasing growth of *Machine Learning* that happened the last two decades, it has become the most popular language for this purpose, due to its focus on *easiness*, its duck typing[5] and its strong Object Orientation: everything can be treated as an object on this language. This is a very interesting feature, as it facilitates features as sharing memory, abstract processes, and much more.

And, of course, it is *Open Source*, so it is always under community improvements, and there are a vast number of incredibly useful third party libraries, which are impossibly easier

[4] https://www.robotnik.es/web/wp-content/uploads/2014/04/TB_robot.pdf

[5] This refers to Python guessing about your code, coming from the phrase *"if it looks like a duck and sounds like a duck, chances are it's a duck."*

to deploy onto your code.

This makes this language a really potential candidate for the applications to develop (and that's precisely the reason that explains its huge growth on the software market).

For our target, we will use Python on its version 2.7. Although it is a relatively old version of the language, it is necessary to mantain the compatibility with ROS (section 3.3) bindings, which have not still taken the leap to the newest major version ($3.x$) on Python.

Nevertheless, we have to mention the fact that Python is an *interpreted* language, which means that its sentences are projected on another program (the Python interpreter, which executes them), and not directly by the processing hardware (CPU/GPU). This can be a handicap, as it makes the code execution much slower, in comparison with standard *compiled* languages, which are run directly as processes, and grabbed by the computer hardware for its execution (as C, C++, Picky, etc.).

3.3 ROS

As it is said on [22], *ROS* (Robot Operating System) is *an open-source, meta-operating system for your robot*, maintained by the *OSRF* (Open Source Robotics Foundation). It is a framework that provides a distributed, easily-scalable environment of *nodes*. These nodes are programs which are independently running on the computer (or distributed over a P2P network), so they can perform individual tasks. However, they can communicate between themselves on a synchronous way (over *services*, implementing a client-server role system between nodes), or on an asynchronous way, via *topics*, which will be the main benefit we will take advantage of. These topics, which rely on a standard TCP/UDP communication between sockets via the `loopback` interface, are intended for an unidirectional, streaming communication, where a node can take a role: *publisher* (if it is writing data inside the topic), or *subscriber* (if it is reading the data that publishers are broadcasting into the topic). The data stream through the topic, however, is not unrestricted. It must follow a ROS specific syntax, the *Message* type, which is strictly defined for the communication purpose (geometry, sensoring, etc.).

For our project we will be using the 2016 *LTS* (Long Term Support) version, called *Kinetic Kame*[6]. This is the version bundled on the installation of JdeRobot[7].

ROS provides libraries and bindings for C++, Lisp, and *Python* (`rospy`). They allow, among plenty of other stuff, to really easily set up a topic between two or more programs, which will be seen as ROS nodes.

However, this topic communication will be abstracted on our project by the `comm` library, as it will be seen on subsection 3.4.3.

ROS also provides a Debian package, called `rosbash`, which allows to, in a very handy

[6] http://wiki.ros.org/kinetic
[7] https://jderobot.org/Installation

```
...
import rospy
from std_msgs.msg import String
...
rospy.init_node('listener')          # Starting the node entity.
rospy.Subscriber('chatter', String)  # Instantiation of the topic subscriber.
rospy.spin()                         # 'Infinite loop' listening to the topic.
...
```

Figure 3.7: Simple stablishment of a listener node through `rospy` (code from [3]).

way, manage nodes and packages from a standard `bash` shell. The most remarkable feature for us is the command `roslaunch`, that launches a ROS node with a certain specific settings, configurable via a `.launch` file (which follows a XML formatting). An example for the file structure can be found on Figure 3.8.

3.3.1 `usb_cam`

ROS node that creates a topic and publishes into it the digital video data incoming from a USB camera, into the topic `/usb_cam/image_raw`.

This node will be used on the first approach to the sensing+actuating node (section 6.7), with the purpose of retrieving images from the Sony EVI D100P camera (Figure 3.1). In consequence, a custom configuration file[8] is required. We can have a glance on that configuration file (Figure 3.8).

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video1" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
</launch>
```

Figure 3.8: Example of `usb_cam-test.launch` configuration file for a ROS node.

Usage: `roslaunch usb_cam-test.launch`

[8] https://github.com/RoboticsURJC-students/2017-tfg-nacho_condes/blob/master/resources/usb_cam-test.launch

3.3.2 `openni2_launch`

This ROS binding [23] provides the launch files for the `rgbd_launch` node. This node publishes on several topics the RGB+D images provided by the Asus Xtion (Figure 3.3), performing at the same time the registration process.

This node will be used on the second iteration of the sensing+actuating node (chapter 6). It will connect the Xtion sensor to the component, providing the real-time imaging through the topic.

Usage: `roslaunch openni2_launch openni2.launch`

3.3.3 `kobuki_node`

This ROS package contains a bunch of launch files. Among them there is the one we will use: `minimal.launch`, which starts the *nodelet*[9] that gives us the total control of the Kobuki robot connected to the computer[10].

This node will be used on the second iteration of the sensing+actuating node (chapter 6). It will connect the Turtlebot motors to the component, providing the topic to command movements to them.

Usage: `roslaunch kobuki_node minimal.launch`

3.4 JdeRobot

As described in section 1.3, JdeRobot[11] is a distributed development platform/middleware, born in [24]. It stands out mainly for two key aspects:

- *Hardware abstraction:* it behaves as an intermediate layer between control software (written by the programmer) and hardware, which can be a real device (a robot, drone, camera, laser scanner, etc.), or a simulation (on the open source world simulator Gazebo[12]). This way, the bidirectional flow (information from sensors, and commands from the computer) is sent the same way, *no matter the kind of the robotic device*.

  As well, this abstraction layer allows various computers to interact simultaneously with the hardware, as the communications are also abstracted to ROS topics or ICE endpoints (it will be properly explained at subsection 3.4.3), where a program has to just listen/talk to be in on. This provides a very valuable *software and hardware scalability to the platform, and to the developed programs*.

[9] A ROS nodelet performs multiple simultaneous processes, and consequently opens several topics.

[10] We refer to the robot as *Kobuki* now because the mobile base of the Turtlebot (Figure 3.6) is the only ROS device which we will control in our application.

[11] https://jderobot.org

[12] http://gazebosim.org/

Let's have a look on a possible example on the Figure 3.9. This could represent an scenario where somebody wants to virtually test a navigation algorithm. Thus, in the *Computer 1*, a reactive controller is running, sensing the environment through a real laser scanner and a RGB camera (as in the work developed at [20]). This controller receives data from the sensors, computes a proper navigation response, and sends it to a virtual robot, simulated on Gazebo.

Additionally, another machine (*Computer 2*) is running a viewer, which allows it to draw the images seen by the camera, and the laser readings from the scanner. So, this component only receives the data from the sensors, and does not send any kind of data to the devices.

We can see that both components can perfectly run together and on different machines, even when they are written over completely different languages (Python and C++, respectively). In addition, we can perfectly handle virtual and real devices simultaneously, even if they talk through different interfaces (ROS or ICE), due to the perfect support of this division by the `comm` (subsection 3.4.3) library.

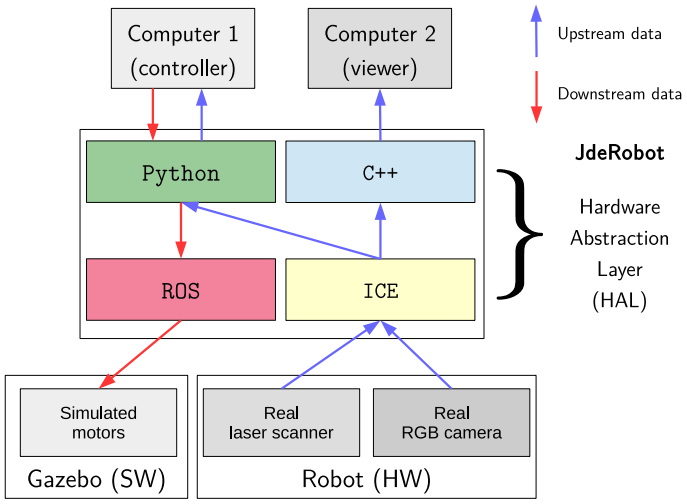


Figure 3.9: JdeRobot abstraction layer, and a possible use distributed, multi-middleware scenario.

So, in this easiness and flexibility resides the main advantage of using JdeRobot.

- *Wide device support:* JdeRobot provides full compatibility with ROS Kinetic Kame, so it can perfectly integrate with ROS Nodes (in our concern, we can communicate with the Turtlebot and the Xtion devices via several topics that the ROS intermediate nodes open).

- *Behavioral based on threaded parallel schemes:* as it is introduced at [24], inside a component, we will find one or more *schemes*, objectified on threads. These threads run concurrently with an specific timing (so it does not overload the CPU in vain if a few

iterations per second are enough for a vivacious and correct response).

These schemes perform different tasks each, as seen on Figure 2.1 on a non-blocking way, and share memory. This has been followed on a comfortable way on our implementation: the threads are independent, but the tasks they control are performed by Python objects, which are interconnected between them:

As an example, we can see how this has been performed inside our Python code:

```
...
cam = Camera(ros_topic)          # Creation of the camera.
net = Network(graph_model)       # Creation of the CNN.

net.setCamera(cam)               # Connection of the camera and the CNN,
                                 # which now can share memory.

t_cam = ThreadCamera(cam)        # Instantiation and start of the thread which
t_cam.start()                    # implements the Camera schema.

t_net = ThreadNetwork(net)       # Instantiation and start of the thread which
t_net.start()                    # implements the Network schema.
...
```

Figure 3.10: Handling schemes on Python with objects and threads.

To sum up, we have described the goodness of the JdeRobot framework for our purposes, and seen how we can implement the *useful schemes paradigm* on an easy way into Python, our development language.

Now, in the next subsections, we will examine which of the JdeRobot components, apart of the structure, have been of greatest interest for us.

3.4.1 Digit Classifier

This JdeRobot component was originally designed by David Pascual [15] and Nuria Oyaga [16], and it was used on this project to land on the concept of neural networks.
In rough outline, its function is to classify on-demand or in real-time (under the mentioned threaded schema structure) the incoming images from a video source, mapping them into digits from 0 to 9. It was written in Keras and Caffe originally, and we have extended its scope to TensorFlow, bundling it all in a specific JdeRobot component[13].

It will be studied in more detail below (chapter 4).

[13] https://github.com/JdeRobot/dl-DigitClassifier

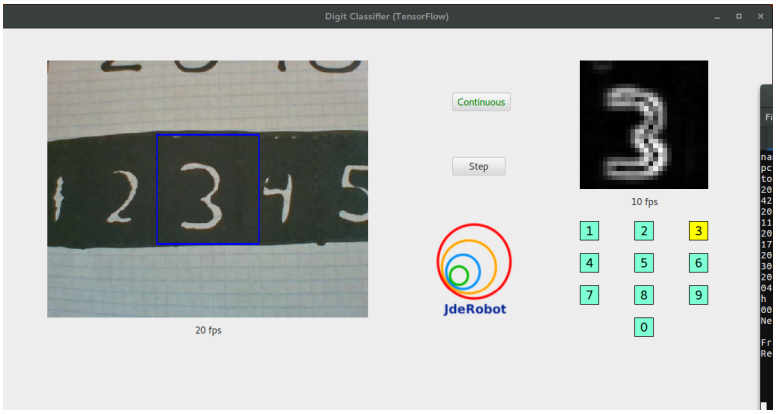


Figure 3.11: `DigitClassifier` on action.

3.4.2 `evicam_driver`

This driver, bundled into JdeRobot[14], allows the user to send movements commands to a Sony EVI D100P camera Figure 3.1) and retrieve information from it, creating an ICE endpoint that is ready to interact with the camera *PT* (Pan, Tilt) motors.

As this is a low-level driver, written in C++, it requires to be used on a specific way, which has been documented[15] to be easily applied in the future.

This driver defines an interaction API with the camera, which allows us to get the values from the motors encoders:

```
import config
import comm
...

cfg = config.load('yml_configuration_file')
jdrc = comm.init(cfg, 'NodeName')


# Instantiation for the motors:
PTMotors = jdrc.getPTMotorsClient('NodeName.PTMotorsEndpoint')

print(PTMotors.getLimits()) # Shows the max/min values for pan, tilt
                            # and each speed.

print(PTMotors.motors.data) # Shows the current values for pan, tilt
                            # and each speed.

# Let's move the camera! As easy as:
PTMotors.setPTMotorsData(new_pan, new_tilt, max_pan_speed, max_tilt_speed)
```

[14] https://github.com/JdeRobot/JdeRobot/tree/master/src/drivers/evicam_driver
[15] https://jderobot.org/Handbook#PanTilt_Teleop

3.4.3 comm

comm is the basic library included on JdeRobot to perform communications between different components. It is what supports all the data flows in a typical scenario (Figure 3.9).

comm consists on a collection of bindings to easily create a link between two components, or between a device and a component. On the lowest level, we can use it relying on ROS (through topics as it was explained before on section 3.3), or through an ICE proxy. ICE[16] is an object-oriented middleware that, in our purpose, allows to abstract a data flow to a TCP/IP endpoint (an address/hostname, and a port), which can even support a communication between two or more programs inside the same machine.

To create a communicator with comm, it needs the specification for that link (underlying middleware, topic/endpoint, etc.), so it uses the JdeRobot standard: YML[17] configuration files, which must follow a similar format to Figure 3.12.

```
DigitClassifier:
  Camera:
    Server: 1  # 0 -> Deactivate, 1 -> Ice, 2 -> ROS
    Proxy: "cameraA:tcp -h localhost -p 9999"
    Format: RGB8
    Topic: "/usb_cam/image_raw"
    Name: cameraA

#  Framework: "Keras"  # Currently supported: "Keras" or "Tensorflow"
#  Model: "Network/Keras/Model/net.h5"  # path to model

  Framework: "TensorFlow"  # Currently supported: "Keras" or "Tensorflow"
  Model: "Network/TensorFlow/mnist-model/"  # path to model

  NodeName: dl-digitclassifier
```

Figure 3.12: YML format required by comm.

In the previous example (3.4.2) we can see an example of an instantiation of a global communicator through comm (which then provides the clients to interact with the device).

3.5 OpenCV

OpenCV (*Open Source Computer Vision*) is a C++/Python/Java open-source library[18] (natively written in C++) for Computer Vision purposes. Among the classic/*state-of-the-art* methods it bundles, we can find functions suitable for face recognition, image stitching, eye movements following, establishing markers for augmented reality, etc.

Its general focus is *efficiency and real-time functionality*, thank to low-level optimizations on the system hardware (i.e. integration with Nvidia CUDA and OpenCL GPU processing libraries). Thus, the excellent performance achieved by this open source library has turned it into the *de facto* standard between every kind of users (from researchers to big companies or

[16] https://zeroc.com/products/ice
[17] Legible data serialization format.
[18] https://opencv.org/

even governmental bodies, as their website stands).

The main benefit we will grab from this library will be mainly for image analysis (such as *Haar Cascade* classifiers, or edge detectors).

3.6 NumPy

NumPy[19] (*Numeric Python*) is a library for Python (written in C++), born to extend the numerical abilities of this language.

It provides a powerful `array` class, which allows to keep a N-dimensional collection of values/objects in a really handy way (in comparison with Python's standard *lists*). It also provides a rich interface to describe the arrays (such as advanced indexing, shaping, data formatting, etc.).
This will such an useful resource on this work, for 3 main reasons:

- *Matrix representation of images:* every processed image is handled as matrices or bigger order tensors (the concept of matrix generalized for any number of dimensions), so visualizing/slicing them becomes a trivial task.

- *Abstract structure to keep objects:* it allows to store different objects in a `np.array` object, providing an advanced API for indexing, and conditional checks to instantly retrieve the elements fulfilling a specific condition.

- *Saving variables into disk:* this is an useful feature for debugging purposes. `np.save()` allows to save any variable (even non-NumPy ones, like dictionaries), finding it on a `.npy` file, ready to be traced and debugged.

In the same way than OpenCV, this is a numerical library widely adopted between Python users. This is due to the easiness of handling of its types and structures, that provides an immediate data exchange format with other parties software.

3.7 TensorFlow

TensorFlow[20], which is the core component of this project, is an open-source software library for high performance numerical computation. It was originally created by the Google Brain team, and it offers an excellent background for *machine learning* tasks.

Its internal functionality is based on *graphs*, composed by nodes which operate and exchange data values establishing *a flow of tensors* (as previously described, a tensor is the general term to describe a multidimensional structure). These tensors can be formed of different data types (images, words, poses, numbers, etc.), which is the key for the versatility it

[19] http://www.numpy.org/
[20] https://www.tensorflow.org/

offers for a large variety of projects[21].

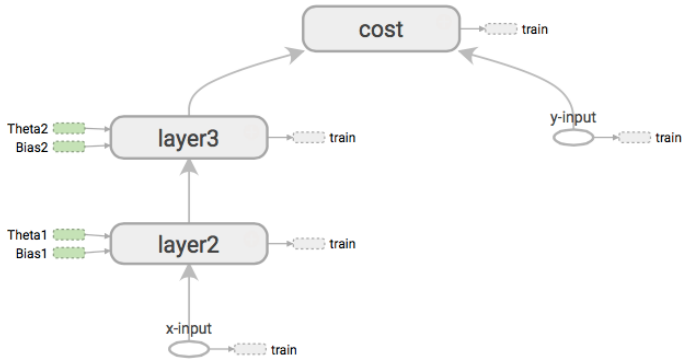


Figure 3.13: Basic graph on TensorFlow (2 convolutional layers fed to a cost function).

All these possibilities make TensorFlow a very optimized ecosystem to implement *deep learning* models (Deep Neural Networks). In addition, it is optimized for parallel GPU hardware. This gives a network the opportunity to experiment a performance boost, since it can reduce significantly the time it takes to make an inference (and even work on a system with a cluster of GPUs, although this is focused to more exigent systems than the one we create on this work). For our

Since it was launched (November 2015), it has been adopted by many big companies which have used TensorFlow as the base for their Artificial Intelligence applications, such as Twitter, Intel, Google, eBay, Xiaomi, Nvidia, etc.

As we will see later in the specific components, it allows to *train* a neural network (for its later use) or even *load* a specific model (kept on a Google Protobuf[22] `.pb` file). This is a really interesting feature, given that we are able to retrieve a bunch of pretrained models and embed them into a generic neural network.

3.8 Keras

As it is stated in [15], Keras is a high-level *neural network library*, written in Python and capable of running on top of either TensorFlow or Theano (another *deep learning* library).

Hence, it is an abstraction with the goal of programming and handling a neural network on an simpler way, relying on a powerful library (treated as *backend*) as TensorFlow to perform all the numeric operations. As well as TensorFlow, it is capable of loading previously compiled and saved models, on the serialization standard HDF5[23] (`.h5` files).

[21] https://github.com/jtoy/awesome-tensorflow
[22] Google's open source mechanism to serialize structured data.
[23] Hierarchical Data Format (v.5): general purpose format to store and manage data.

In our project, support has been provided to use this framework (selecting it on the YML configuration file), although our main interest has been TensorFlow due to the significative difference of processing speed between both frameworks (being TensorFlow twice as fast as Keras).

3.9 PyQt

Qt[24] is an cross-platform object-oriented framework for building GUIs (*Graphical User Interfaces*), originally developed by a Nokia department. It is distributed under a commercial license, although it has a standard GPL license for open-source projects.

A third party company (RiverBank Computing) developed PyQt, a set of Python bindings to interact with Qt (originally written in C++). It is structured in units called *Widgets*, which contain blocks (*Labels*).

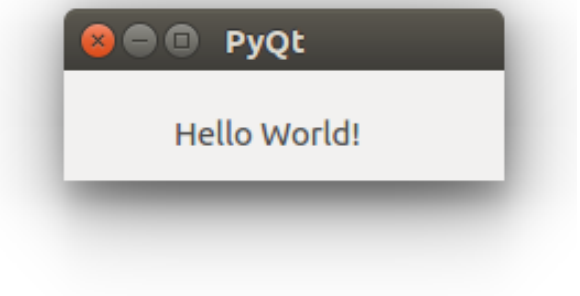
All this allows to easily deploy a GUI-based(*Graphic User Interface*) program:

```
import sys
from PyQt5 import QtGui, QtWidgets

def window():
   app = QtWidgets.QApplication(sys.argv)
   w = QtWidgets.QWidget()
   b = QtWidgets.QLabel(w)
   b.setText("Hello World!")
   w.setGeometry(100,100,200,50)
   b.move(50,20)
   w.setWindowTitle("PyQt")
   w.show()
   sys.exit(app.exec_())

if __name__ == '__main__':
   window()
```

(a) *Hello world* example code.



(b) Resulting window.

Figure 3.14: Example of a *Hello World* window with PyQt5 bindings.

As it is the last available version at the time this is developed, we will use the version 5 of Qt. Hence, our binding library is `PyQt5`.

[24] https://www.qt.io/

3.10 Threading

`Threading`[25] is a Python standard library which offer a high-level API for threading processes. This means to run programs on more than one single job for the CPU, which allows to be capable of perform several tasks on a simultaneous way.

This is very convenient for our purpose, as we want to stick to a multiprocessing paradigm (Figure 2.1). So, this way we can have dedicated threads to grab the new camera images, update the GUI, and make the Neural Network to load new inferences on the last image detected.

The `threading` provides a generic class for a thread. Our only task is to create a custom class which inherits it, customizing the `__init__` and `run()` methods our own way:

```
...
import threading
from datetime import datetime
...

class MyThread(threading.Thread):

   def __init__(self, foo, bar):
      '''
      This is the method which will be called at the creation of the thread.
      '''
      self.my_foo = foo
      self.my_bar = bar
      self.time_cycle = 100 # ms
      threading.Thread.__init__(self) # Rest of the initialization.


   def run(self):
      '''
      This is the task the thread will perform once.
      If we put an infinite loop inside, we have a periodic thread.
      '''
      while True:
         start_time = datetime.now()
         # Grab an image, or run an inference on the neural network...
         self.my_foo.doMyStuff(self.my_bar)
         end_time = datetime.now()
         dt = end_time - start_time

         # If it did not take the refresh time, it sleeps until it arrives.
         if dt < self.t_cycle:
            sleep(self.t_cycle - dt)
```

In addition, as we can see, it is possible to control the update period of the thread, so we

[25] https://docs.python.org/2/library/threading.html

can decide how much time it will be elapsed between two consecutive executions of the thread task (and of course this is a tunable parameter).

Chapter 4

`DigitClassifier` node

4.1 Description

This JdeRobot component was originally designed by David Pascual [15] and Nuria Oyaga [16], and it was used on this project to land on the concept of neural networks.

Its design aims to *classify handwritten numbers* with the use of a Convolutional Neural Network (subsection 1.2.5), which classifies the incoming images from a video source.

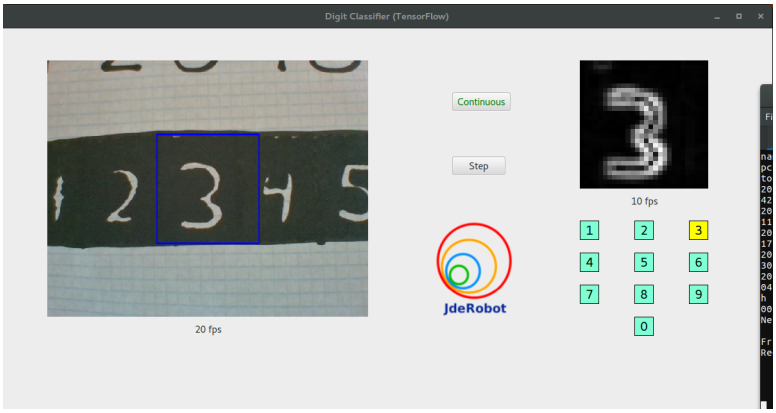


Figure 4.1: `DigitClassifier` on action.

This kind of application (handwritten digits classification) is the typical first milestone to achieve in the domain of deep learning concepts and neural networks architectures, so lots of documentation are available online (a basic domain of the framework can be successfully achieved on the official TensorFlow tutorials[1] which, as said, teach how to deploy a handwritten digit classification system).

As we mentioned, previous existing implementations were written in Keras [15] and Caffe [16] (Python libraries to implement Deep Learning algorithms), so we made the same on TensorFlow (section 3.7), to accomplish an initial domain of this Machine Learning framework.

[1] https://www.tensorflow.org/tutorials/layers

After this, Keras and TensorFlow versions were merged on a single JdeRobot component: `dl-digitclassifier`[2], which has the scope to be functional on both libraries and, in addition, supports image streams from ICE endpoints or ROS topics (thanks to the `comm` library).

As it can be seen it the *README* file[3] in the repository, configuring the component is made pretty simple thanks to the YML file, so the reader is asked to feel encouraged to give it a try.

4.2 Node architecture

As specified in Figure 2.1, we divide the entire node functionality into individual tasks. These tasks have been implemented invoking a dedicated thread (using the `threading` library) for each one.

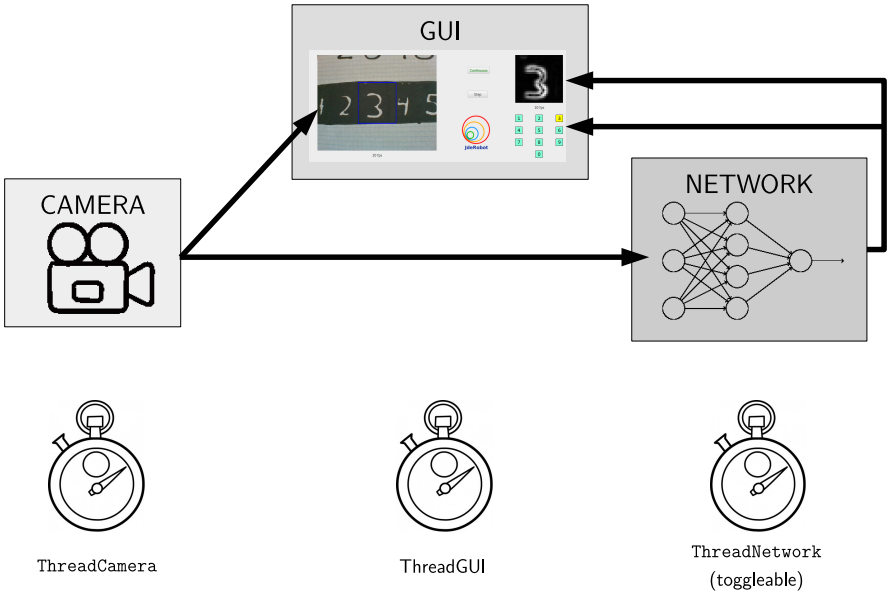


Figure 4.2: Infrastructure of the component (3 threads).

This establishes the first functional difference with the previous version, which was powered by only 2 threads (for the camera and the GUI), performing on-demand (blocking) inferences on the neural network. Implementing a new thread dedicated to the neural network allows to have an instantaneous output digit from the network, ready to be grabbed on an easy and non-blocking way.

Using the specified `threading` library, this deployment is as easy as: Nevertheless, care

[2] https://github.com/JdeRobot/dl-digitclassifier
[3] https://github.com/JdeRobot/dl-digitclassifier/blob/master/README.md

```
import Camera, ThreadCamera
import GUI, ThreadGUI
import Network, ThreadNetwork
...


# We instantiate each object with its pertinent thread...
cam = Camera(cam_parameters)
t_cam = ThreadCamera(cam)

gui = GUI(gui_parameters)
t_gui = ThreadGUI(gui)

net = Network(net_parameters)
t_net = ThreadNetwork(net)

# Communicate them (according to the scheme)...
net.setCamera(cam)
gui.setCamera(cam)
gui.setNetwork(network)

# And start the application!
t_cam.start()
t_gui.start()
t_net.start()

gui.show()
```

Figure 4.3: Schematic code to instantiate the components of the node.

has to be put on respecting parallelism between threads. This infrastructure is optimum for asynchronism, which means that one component does not depend on any others (e.g. the GUI component does not need the network to finish the next inference and return the classified digit in order to refresh the interface: it grabs the last inference made by the network *from the own network*, which will be automatically updated when a newer one is available).

This implies a respect to the independence between threads. It is the key for an asynchronous behavioral, as we want the component to yield the best possible performance.

4.3 Image processing

For every digit recognition system using a CNN, the approach is the same: *processing a 28×28 px image*[4] containing a binary image of the digit itself. In addition to this, our system aims to a robust and correct classification at every feasible situation. Handwritten digits can often be found in visually harsh environments (poor quality sensor or lighting, trace indistinguishable

[4]This particular shape is due to the used dataset, as it will be seen later.

of the background, etc.). Given this, before inserting an image on the network, we have to subject it to a *preprocessing* pipeline:

1. Grab the central region of the incoming image from the camera (blue square on Figure 4.1).

2. Convert the image to grayscale, as the color information is not relevant (hence, the network does not deal with RGB images, but just a grayscale 28×28 matrix).

3. Soften the image with a *gaussian blur*, in order to reduce the possible noise on the image. We convolve the image with a square kernel of size 5 px.

4. Resize the image to the necessary shape of 28×28 pixels.

5. Lastly, we *extract the edges* of the image. This is not trivial, as the information we want to observe for the digits are the *edges*. Otherwise, images on different contrast conditions would not be supported, as that would imply very different weights on the network for each situation. For this edge extraction, we apply a Sobel operator (which can be seen like computing the directional gradient/derivative in the image, as stated in [25]) on each direction (vertical/horizontal), in order to detect the existing edges, and add them together in a single edge image, which is the final output of the pipeline.

As a result, we obtain a softened shape map (even from different images, as seen on Figure 4.4), which is a good start for the neural network to infer. This pipeline will be applied to every image (in the training process and while performing a new inference).

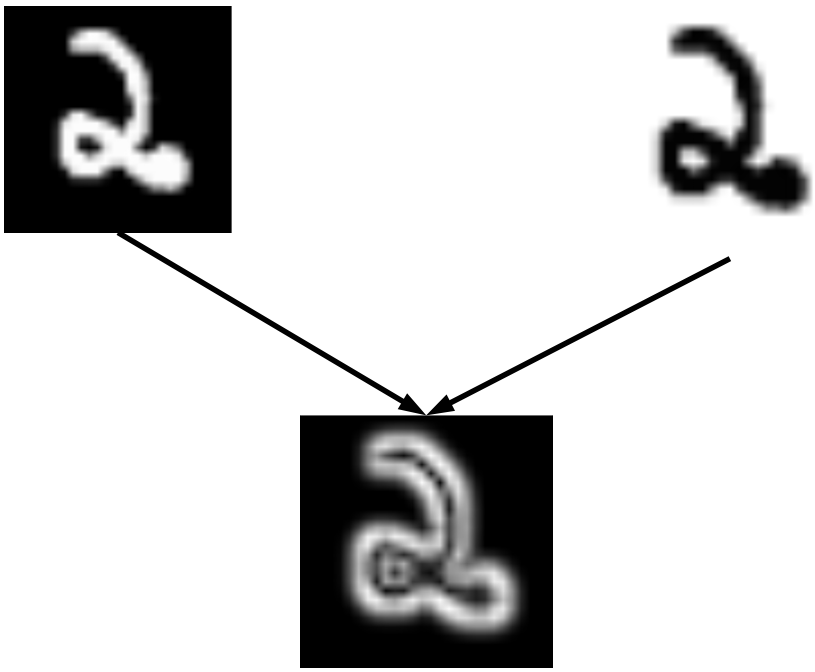


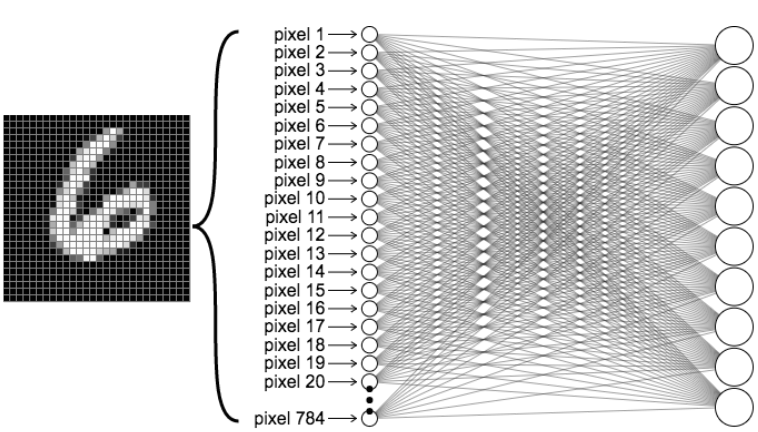
Figure 4.4: Result of the preprocessing (identical for both images).

4.4 Digit classification CNN

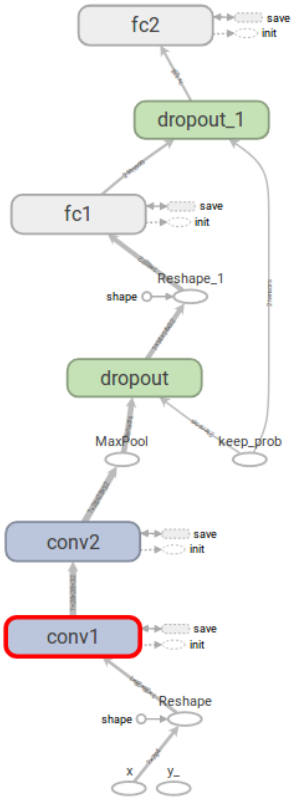
The implementation of this convolutional neural network consists of a concatenation of layers, following the scheme shown on Figure 4.5. These layers are disposed in a serial structure, so the output of one layer (a vector containing what each neural activation function yields) acts as the input for the next one.

For this reason, we need a first input layer, where all the pixels of the input image are mapped to a input neuron, on a bijective way.

As stated before, the input shape for the images is 28×28 px (a total of $28^2 = 784$ px), so we firstly perform a *reshape* operation, which arranges the pixels on a 1-dimensional array of pixels. That will be the input for the network.



(a) High level layers visualization.



(b) Low level layers visualization.

Figure 4.5: Model of the implemented CNN for our system.

1. `conv1`: first convolutional layer. As described in subsection 1.2.5, it performs a 2D convolution between a $5px \times 5px$ square mask/kernel (`W_conv1`), and the image (which is seen again as a 28×28 matrix to perform the convolution). Later, the layer adds a bias/intercept term (`b_conv1`). Thus, we obtain the activation for each neuron (the ReLU operator applied to the local convolution in the environment of that particular pixel), `h_conv1`.

```
# Illustrative purposes.
# Some additional parameters (padding, stride) ignored.
h_conv1 = tf.nn.relu(tf.nn.conv2d(x, w_conv1) + b_conv1)
```

To get a better understanding of what is happening here, we can have a glance of the weights learned on each neuron, for each digit (as we will study further below), on the Figure 4.6.

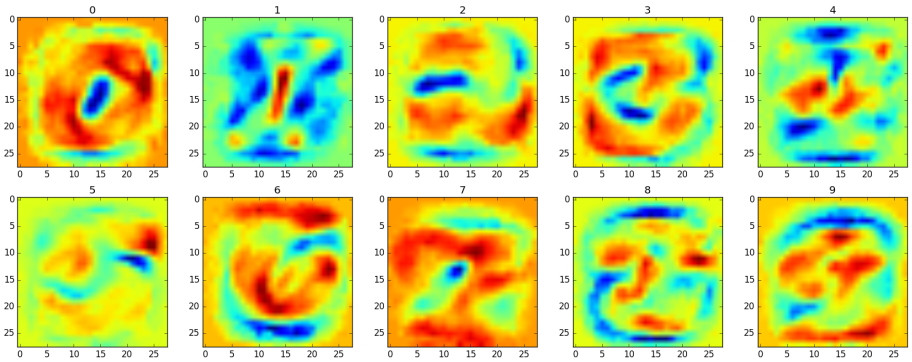


Figure 4.6: Heatmap for the learned weigths for each pixel and labeled digit.

This can begin to illustrate what is happening here: each neuron has a set of 10 tunable kernels, which depend on the value of what it has seen during the training process on its corresponding pixel and its environment. This illustration belongs to a simplified version of the network, where a convolution is not performed, but a simple *matrix multiplication* (which can be seen as a convolution of kernel size equal to 1). So, we can see that the network is just learning where typically enabled pixels are situated on the input for each possible label ($0 - 9$).

2. `conv2`: second convolutional layer. It performs the same operation taking the output from the previous layer as an input, using a different weights mask and bias terms.

```
h_conv2 = tf.nn.relu(tf.nn.conv2d(h_conv1, w_conv2) + b_conv2)
```

As we stated before, the tensor which will be convolved with the weights of this layer (`w_conv2`) is now the output of the previous layer (`h_conv1`).

So far, what we have done is extracting patterns on each digit type (e.g. discovering typical circles on 0 and 8, which are always present on the same zone of the image).

3. `pooling`: as the activation maps can be growing in size as we perform feed forward propagation, a *pooling* operation is performed. It consists of spatially downsampling its input (the previous layer activation map). Concretely, we retain the *maximum* value for each 2 pixels, which is known as 2x2 `max pooling` (Figure 4.7).

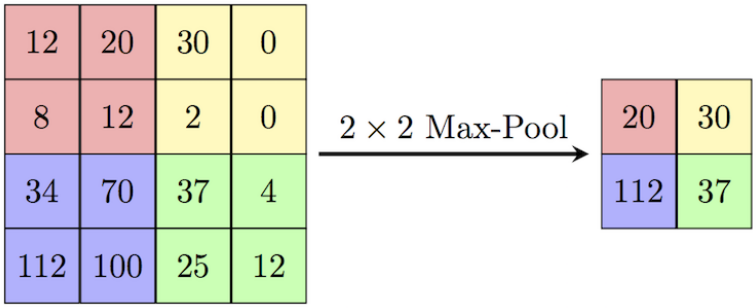


Figure 4.7: Max-pooling operation on a matrix.

```
# Some additional parameters ignored here
# (kernel size, strides, padding).
h_pool = tf.nn.max_pool(h_conv2)
```

4. `dropout`: this layer does not strictly perform any mathematical operations. It lets pass the tensors through it, but randomly switching off some neurons. This is parameterized by a user input, using a variable called `keep_prob` (which stands for the probability of a neuron staying switched on). In our case, we set it to 0.5 (50%) during the training process, to avoid overfitting by forcing the network to modify the neural paths randomly, as not every neuron is available on every moment. This is kind of similar to augmenting the dataset during the training process. The rest of the time (when the network is used to make inferences), this parameter is set to 1.0 (100%), which means that no neurons are switched off at all.

```
# The value for keep_prob is parameterized.
h_drop1 = tf.nn.dropout(h_pool, keep_prob)
```

5. `fc1`: first fully connected layer. So far, we have seen the pipeline as operable matrices. From now on, we go back to the *single-dimension* activation map (array of outputs) model.

   These layers, also known as *dense* layers, are distinguised because every neuron is connected to every activation from the previous layer. So, this kind of layers are used for *pattern association with labels*, due to the relationship they can infer between every input.

   A thumb rule on this kind of layers is the more neurons included, the better generalization (the more correlation patterns we can detect between the activated zones on the incoming activation map). So, our implementation establishes a first dense layer of $14 \cdot 14 \cdot 32 = 6272$ neurons.

```
# Firstly, the last activation map is flattened into a 1 dimension array.
h_pool_flat = tf.reshape(h_drop1, [-1, 14*14*32])
```

```
# We model the full connection as a matricial multiplication
# (with a weight size of 14*14*32)
h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, w_fc1) + b_fc1)
```

6. `dropout`: to gain in strength, we deploy another dropout layer (which switches off random neurons with the same probability than the first dropout layer). This is remarkable, as we had performed a dropout process on the *feature retrieval phase (first layers)*, but not on the *pattern searching phase (dense layers)*.

```
h_drop2 = tf.nn.dropout(h_fc1, keep_prob)
```

7. `fc2`: the output layer. It connects all the outputs of the previous dense layer and groups the output in a 10-dimensional vector, which contains the obtained *logits*, a representative number for that class on the given image, which can be interpreted as a linear *reward/tendence* [26] to belong to that class.

   As the last step, it applies a *softmax* (σ) function to the output:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{4.1}$$

   This function normalizes the output, mapping it between 0 and 1. This converts the mentioned raw numerical output into a *probability* of being the given class (where the sum of the probability of all possible classes converges to 1).

```
class_tendences = tf.nn.relu(tf.matmul(h_drop2, w_fc2) + b_fc2)

# Output (this is the called node to get the total inference output):
y = tf.nn.softmax(class_tendences)
```

This way, the total output of the CNN is a vector containing the probability of the image belonging to each class. If we keep the argument(s) of the maxima (`argmax`), we can find the most suitable class for that image.

At this point, we can remember that the images entering into the network have been preprocessing, looking for the edges, so the learning process and the weights used through all the recently described pipeline can be generalized to all kind of images (as we will pass it previously through the Sobel edge detection filter).

4.4.1 Training the network

The reviewed pipeline is performed on an image, since it enters on the neural network until it goes out, passing through all the defined layers performing what is called *feed-forward* propagation. However, the values of the activation maps and, hence, of the dense layer outputs (which determine the resulting class) depend on a relatively huge number of neurons, each one with its own weights and biases (for each class). As we can see, that is a ridiculous

number of parameters to tune manually, hence we tune it automatically performing what is called *backpropagation*.

This backpropagation algorithm [27] performs a standard feedforward propagation (as described before) on what we call *training set* (labeled images that will be used *exclusively* for this purpose), which the system takes as examples which to learn of. Then, it compares the obtained output class ($argmax(softmax())$) for each input, and adjusts all the weights of the network, seeking to ensure that the difference between the *desired* output (the *ground truth* labels) and the *obtained* result is minimum. This difference is computed and represented by a value which we call *loss/cost* (the higher its value is, the worse, as there is a bigger difference between the correct and the obtained output).

This tuning process is performed using what is called an *optimizer*, which is an algorithm to search the optimum direction and magnitude update for every weight present on the network, depending on the obtained differences. As this can be such a complex process, it is performed on an *iterative* way: evaluate a batch[5] of training images, obtain the value for its loss, compute the suitable update values, and perform the weights update (scaled by what is called *learning rate*, which determines the magnitude of the leap for each update). This way, after a number of iterations[6], the model should converge to a global minimum for the loss function, which hopefully means that a suitable value for each weight has been found. This is the mechanism responsible of fitting the weights, so thank to it, we obtained, among many others, a reasonable values for the input layer on this network (Figure 4.6).

On our application, the chosen function to compute the lost value is the *softmax cross-entropy* (as the name indicates, it is based on the *softmax* version of the output, which means that it works with probabilities). It stands for the difference between the correct probability for a particular class (remember that each class is represented by a neural unit on the output layer: 1 if it's the suitable class, 0 otherwise), and the obtained output *probability* [28].

This value is obtained on this way, being y the binary correct output, and p the computed softmaxed probability:

$$-(y \cdot log(p)) + (1 - y) \cdot log(1 - p) \tag{4.2}$$

With respect to the optimization algorithm, the most intuitive option is the *gradient descent algorithm* (that, in fact, is perfectly implementable on this system). Instead, we use the *Adam* algorithm, which is an extension of the gradient descent approach. Its advantage is that it *automatically adapts the learning rate* of the parameters adjustment, by computing its moving averages and variances (*momentums*) [29]. This is some more computationally expensive, but it is affordable. In exchange, we obtain an automatic training hyperparameters tuning, which is reflected in a faster and finer convergence to a correct value. It takes as input the cost function, and struggles to iteratively minimize it.

[5] A *batch is a set of examples, which are introduced to the network on a stacked way.*

[6] This necessary number can not be determined, as it depends hugely on the dataset, purpose, network structure, learning parameters, etc. The best option is to monitor the loss/accuracy values during the training process, and stop it when these values converge (otherwise it could result in *overfitting*, that is not convenient as it will only work fine on already seen examples).

Putting all this together, we can build these nodes on TensorFlow on an easy way, gearing the output (y) to the new optimization blocks:

```
# Cost function:
cross_entropy = tf.reduce_mean(
                tf.nn.softmax_cross_entropy_with_logits_v2(labels=self.y_,
                                                           logits=self.y))


# Optimizer (this is the called node during the training process):
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

4.4.2 MNIST dataset

MNIST (*Modified National Institute of Standards and Technology database*)[7] is a public database, which contains labeled binary images of handwritten digits. It is the result of jointing and shuffling[8] two previous NIST's special databases (SD-1, containing digits written by Census Bureau employees, and SD-3, containing digits written by students), which were originally designed as training and test sets, respectively.

All the included images respect the same format, with two key aspects:

- The images are *square* 28×28 pixels matrices, containing the whole number (and anything else).
- The pixels represent binary values. This means that a pixel can only take values as *digit* or as *background*.

Given this, there are two image sets at our disposal: $60,000$ images for training, and $10,000$ images for testing[9]. Among all of them, we can obtain random examples (Figure 4.8a).

This is the image source we use to train our component, remembering that every image is preprocessed (Figure 4.4) before going into the network.

4.4.3 Dataset augmentation

David Pascual [15] and Nuria Oyaga [16] performed what is called a *data augmentation* over the MNIST dataset, with the objective of *pretend* to have a bigger number of samples to train.

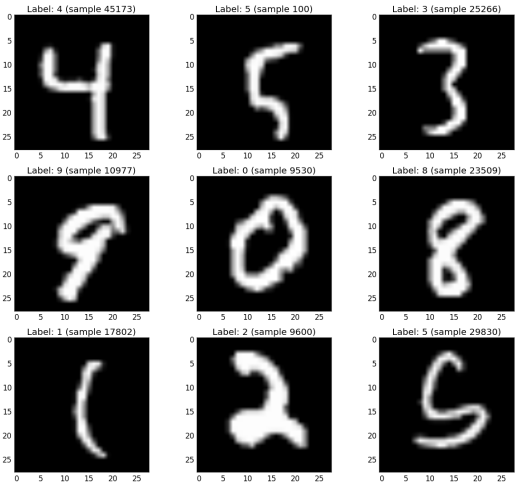
[7] http://yann.lecun.com/exdb/mnist/

[8] This is an important step to perform on databases, because we have to be sure of a correct *generalization*: the data used for training must be of the same kind than for testing, to obtain fair results independently of the chosen dataset on each step.

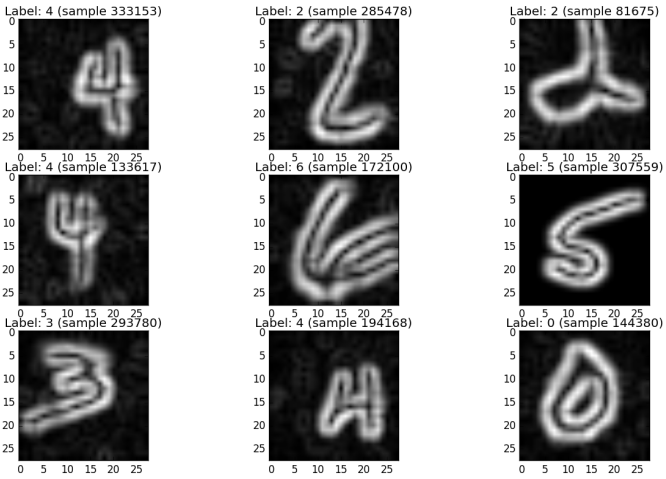
[9] The *test* set must not be used on the training process under any circumstance, to avoid unfair results, as this would be the equivalent to know an exam questions before taking it.

It consists of taking the existing images (on the standard MNIST dataset) and apply to it random (although controlled) transformations: each image suffers *translations, rotations and zooms*, in addition to *gaussian noise*. This has a double purpose: to *get a bigger number of samples* (which is always a good thing), and to *make a better model to process real world images*. As this network will classify real incoming digits from a camera, these digits will have most probably suffered noise (because of the camera and light conditions) and various geometrical transformations (as it can be shown to the camera with a slight tilt, or on a different background, etc.). Thus, we need to create a neural network which is ready to deal these harsh conditions. That is the main reason to augment the dataset performing these transformations.

Taking all this on account, we have at our disposal the datasets they created[10]. As we can see on Figure 4.8b, this set of images is much harsher than the standard one (Figure 4.8a). This will allow to create a much more robust model to classify real world digit images.



(a) Standard MNIST dataset.



(b) Augmented MNIST dataset.

Figure 4.8: Set of 9 random images extracted from the datasets.

These augmented datasets can be combined in different proportions (how many modified images are created for each standard image), which is typically indicated at its name: `training set x-y` means that there are `y` modified images for each `x` original one. To train the definitive model implemented on the classifier, we use the `1-6` model, which means that there are 6 modified versions of an image vs. 1 copy of the original one.

[10] A link to download them for future usages is available on the *README* file of the repository: https://github.com/JdeRobot/dl-digitclassifier.

Chapter 5

`ObjectDetector` node

5.1 Description

Once we have some initial knowledge on CNNs applied to image processing, thank to the `DigitClassifier` component, we develop a new deep learning component, `ObjectDetector`, which is capable of *detect objects in a real-time image stream*[1].

This component has a generic functionality, as the real-time processing capability is only used to display on the image where the objects are, wrapping them on what is called *bounding boxes*. These boxes are the rectangles inside of which, theoretically the detected object is contained, and are delivered along the *detection score*, and the corresponding *class* detected. We will go further on this later.

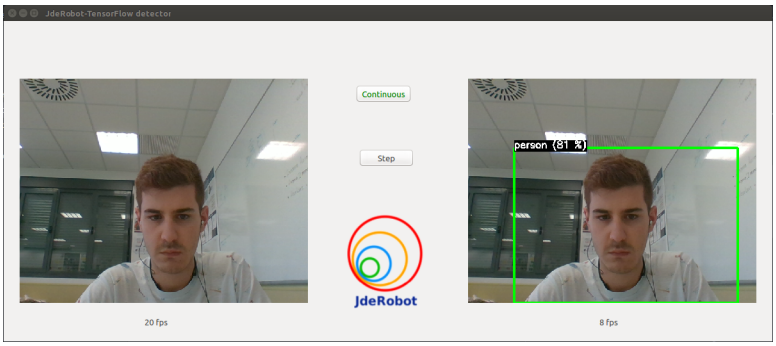


Figure 5.1: `ObjectDetector` working.

We can notice that the type of problem that we are trying to solve now is different. Until now, we were trying to *classify* images, based on a certain features that the network extracted by itself (the response always a classification, inferring an output even when no digit was shown to the camera, like shown on Figure 5.2). Now, we are focused on *detecting* objects inside an image. In other words, we want to distinguish whether an object (of a specific class or type) is present or not in the image that is being currently seen.

[1]For a better compatibility with future usages for this component, it is able to process images from a local video/webcam directly using OpenCV (so `comm` is not required for the execution), specifying this in the YML configuration file.

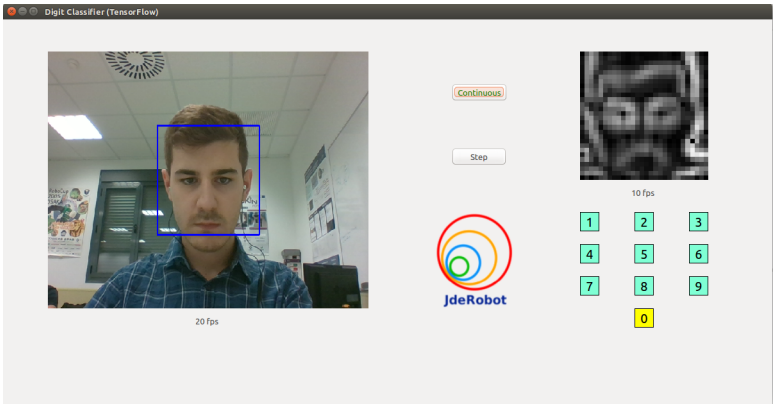


Figure 5.2: `DigitClassifier`: a digit was always returned (or a subtle way of a computer to call you waste of space).

It should be mentioned that this purpose requires a much more complex CNN, which leads to a significantly heavier computational load on the machine (and, as a consequence, an importantly bigger inference time). Hence, a GPU-version of TensorFlow is highly recommended, as the CPU-version would take way too long for a real-time operation system.

Due to this mentioned complexity, *training a detection CNN* is out of the scope of this project (even so, with the proper resources, we could do it with some open-access labeled image datasets). Instead of this, we will pick some public models that the TensorFlow team has made available on the *Detection Model Zoo*[2] (Figure 5.3). On a parallel way, we have created a public mirror repository[3] inside JdeRobot containing some useful found models (for TensorFlow and Keras respectively).

These models have been trained on first-tier hardware, and are materialized in serialized files containing the graph structure and the corresponding weights for each neuron, on a ProtoBuf format (as we stated on section 3.7). These model files are *loadable at runtime* on a TensorFlow graph instance, so invoking one of these imported graphs is an easy task:

This way, we have successfully loaded the graph with its weights on the TensorFlow backend. The Python class (`DetectionNetwork`) we have defined allows to do this, choosing the desired model and dataset[4] (we don't want the network to label a person as a dog) through the global YML configuration file.

Another thing to mention is the fact that, defining a `Writer` on the load process, we can inspect the graph structure on TensorBoard, as we will do later.

COCO-trained models {#coco-models}

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes

Figure 5.3: Some available models (July 2018) with their respective performance indicators.

```
detection_graph = tf.Graph() # New graph instance.

# We use a context generator to set this graph as the default one
# on the TF backend:
with detection_graph.as_default():
   graph_def = tf.GraphDef()
   graph_def.ParseFromString(model_file)
   # We load this definition into the backend
   # (which contains the Graph instance).
   tf.import_graph_def(graph_def)
```

Figure 5.4: Generic model loading process.

5.2 Node architecture

This node inherits the architecture developed for the *classification node*, consisting on 3 parallel threads. These threads drive the behavioral of separated objects, committed to perform independent tasks:

- *Camera*: grabs the current image delivered by the communication framework (`comm` or OpenCV).
- *GUI*: launches and updates the interface, with the fresh image from the camera. That image is drawn twice, as one of the representation (left) corresponds to the raw image

[2] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

[3] http://jderobot.org/store/deeplearning-networks/

[4] Compatible with COCO, Kitti, OID and Pascal VOC datasets.

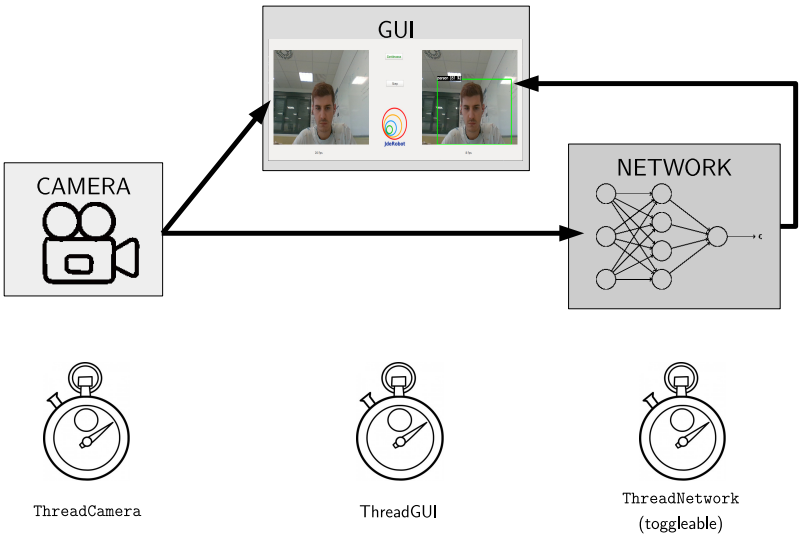


Figure 5.5: Infrastructure of the component (3 threads).

from the camera, and the other one (right) is modified, including the bounding boxes, class and score for each detected object. This mentioned information is taken from the neural network. Hence, this component is connected respectively to the image source and the neural network.

- *Network*: infers continuously the detected objects from the last received image, on an asynchronous way. When an inference is completed, the result (a set for each detection of bounding box, class and score for each single detection) is stored inside the `network` element. When the GUI needs the latest inference data, it just takes that data without any blocking call nor interrupting any process. This complies with the stated asynchronism requirement.

Given this pretty identical structure to the `DigitClassifier` node, the schematic code to instantiate the program is the same than `DigitClassifier`'s (Figure 4.3).

5.3 Detection CNN: SSD

As we have just mentioned, our application uses a SSD (*Single Shot Multibox Detector*) CNN to perform the detection task. This choice has fundamentally taken based on the real-time performance expected from the component. We want it to make inferences as fast as possible (with a reasonable precision), so we chose this kind of architecture (it's more, it yields as good results as its slower counterparts).

The high speed of inference of this kind of detectors is explained with the fact that *it performs a single feed-forward pass* of the image through the network (on a *single shot*, as it name states). According to its official release [4], the rest of *state-of-the-art* detection CNN

technologies approach performing *feature scaling* and *bounding box proposals*. These techniques require more than one pass of the image or, at least, more than one single architecture, which rises the inference time.

5.3.1 Architecture

As we can observe in Figure 5.6, a *SSD* detector has a defined network architecture, with some key aspects to keep its performance vs. inference time on the highest possible value. An inspection of the architecture using TensorBoard (Figure 5.6) reveals the pipeline structure, which is described below.

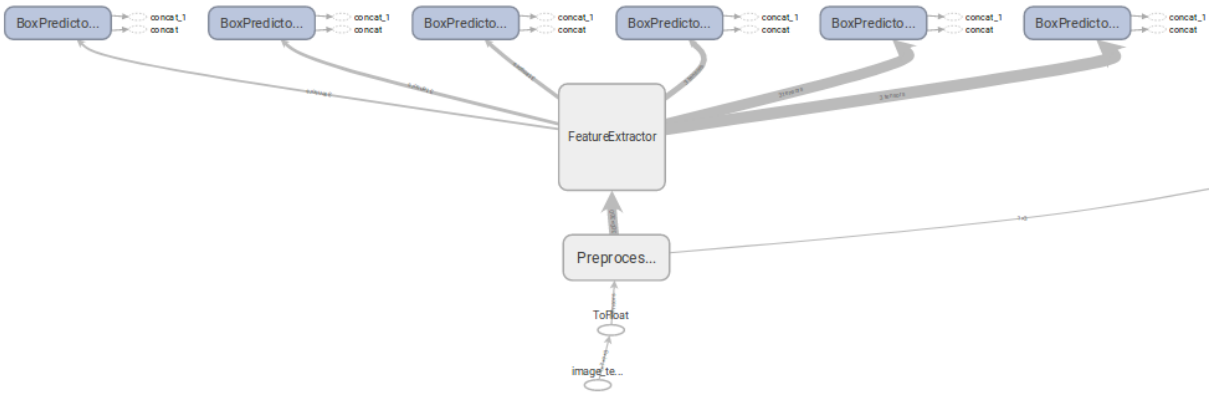


Figure 5.6: SSD architecture on our model.

1. *Preprocessing*: a first *reshaping* stage is necessary due to the SSD operation. We can mention that this reshape is performed *inside* the neural network, working directly with tensors on the GPU, which is probably faster than if we performed it with higher-level operations. This block reshapes the image to a 300×300 size, which is the most typical image size on an SSD detector.

2. *Feature Extractor*: the architecture on a SSD CNN is based on a first group of layers (typically called the *base network*), which deals with the *feature extraction* part (as on the first stage of the classification network we designed on section 4.4). This particular model has a *MobileNet* based feature extraction network[5]. As it is formed by a concatenation of convolutional layers (Figure 5.7), the resulting *feature maps* will be gradually smaller and deeper while we go forward. These maps sets will be convolved with the original image, so the smaller the activation map is, the bigger its *receptive field* will be (hence, it will useful to detect bigger objects). Given this, we are capable of *extract* 6 intermediate sets (described on Table 5.1), with the objective of *detect objects of different sizes*.

3. *Box Predictors*: later, for each extracted set, a dedicated operation is performed (that's the reason why we have 6 identical boxes on the TensorBoard analysis (Figure 5.6).

[5] *MobileNets* are originally designed to embed *small, low latency* deep learning systems on low-spec devices [30], squeezing the trade-off between performance and inference time. So, we can reuse that part of its architecture for our proposal.

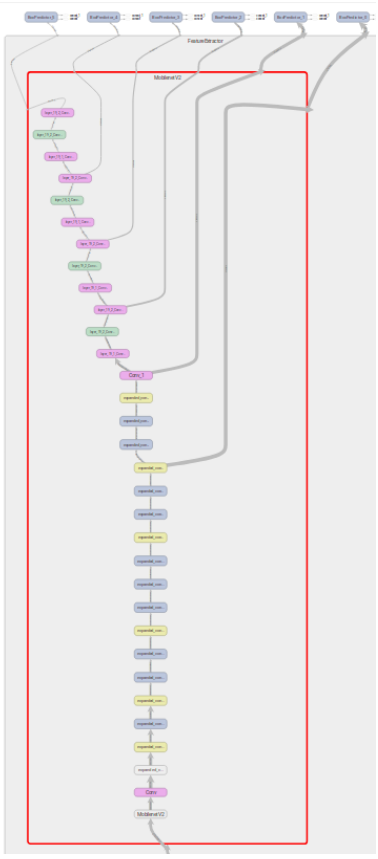
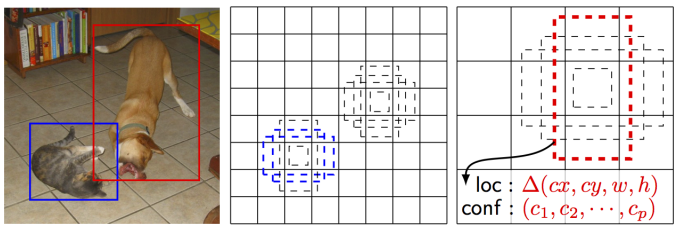


Figure 5.7: *MobileNet* pipeline.

They will perform the same operation but in patches of different sizes/depths, to detect *objects on different scales*). Into this component, for each layer of the extracted feature set, a small set (3-4 typically) of bounding boxes (called *priors*) with different *aspect ratios* are generated (Figure 5.8).



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 5.8: A set of boxes are generated centered on each point of every feature map [4].

After this, these *priors* are convolved with small filters (one per depth channel), which outputs *softmaxed confidence values for each known class*, and *offsets/adjustments for the generated bounding box*. So, for each detected object (on that scale), we know the score for each class and its estimated position inside the feature map.

4. *Postprocessor*: this element does not appear in Figure 5.6, it had to be cropped on the image for geometrical reasons, but it is present on the network structure. It combines the output of all the 6 *Box Predictors* (which contain detections for each feature map

# Set	Shape	Depth
1	1×1	128
2	2×2	256
3	3×3	256
4	5×5	512
5	10×10	1280
6	19×19	576

Table 5.1: Description of the 6 extracted feature maps sets on our implementation.

set), and applies a *Non-Maximum Suppresor*, which only retains the most confident detections, and scales its position to the original image size.

This way, we have a system that, for each introduced image, returns a collection of:

- *Classes*: the detected class (person, cell phone, airplane, dog...) inferred.
- *Scores*: the confidence $\in [0, 1]$ the network has on each object belonging to the decided class (which was the most probable one while the detection was performed).
- *Boxes*: the coordinates of the *bounding box* which wraps the detected object, typically expressed as the coordinates of its top-left and bottom-right corners.

As this is suitable for real-time performance, we are now capable, on a standard computer hardware, to feed it with a video streaming (a file, or live camera images).

5.3.2 Importing a pretrained model

We have mentioned the training process of this kind of CNNs. It is not complicated from the point of view of the entire system (*black box*), it only needs a dataset containing images with classes and *ground truth boxes*[6] (Microsoft's COCO, Pascal VOC, etc.). With these images, the SSD architecture learns to perform a better *regression*, to achieve a better fitting of the boxes with respect to the original ones, evaluating it using the *Jaccard similarity coefficient*, or IoU (*Intersection over Union*). This measure evaluates how good the overlap between the true and the estimated boxes is (Figure 5.9). Additionally, it performs a standard *back-propagation* process similar to the one we executed on the previous component (subsection 4.4.1).

The much heavier complexity of this task is not our point where to focus, so we will *embed pretrained models*, available thank to the TensorFlow team on the previously mentioned *Detection Model Zoo*[7].

In our case, we have selected a *SSD detector*, with a *MobileNet v2* feature extraction base network, trained on the *COCO* dataset [8]. As this dataset support 90 object types (*person, dog, airplane, toothbrush, apple, cellphone, etc.*, those are the classes we are able to detect.

[6] *Ground truth*: what the network knows to trust, what is told it to be the true position of the object to detect.

[7] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

[8] http://cocodataset.org

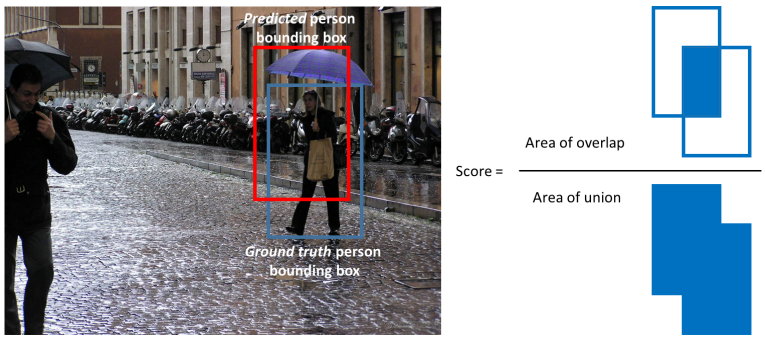


Figure 5.9: *Jaccard similarity coefficient* on a detection (performance indicator used for training a detector).

It provides a lightweight structure which performs inferences at a framerate of approximately 14 fps (on the currently available hardware).

We load this network architecture and weights on the mentioned manner (Figure 5.4).

5.3.3 Network output

As we have specified, the detection network yields detected *classes*, *scores* and *bounding boxes*. Respecting the required asynchronous behavioral, this data is deposited on an accessible zone for the GUI, allowing to begin another iteration when requested.

Therefore, as described at section 5.2 the GUI instance grabs the last detection data the network left on the shared placeholder, and draws it on the visible user interface (Figure 5.10).

5.4 Experiment: testing different architectures

The implemented generic class (`DetectionNetwork`) allows to load on runtime a pretrained TensorFlow model of neural network. We can take advantage of this to perform a quantitative timing benchmark of different architectures and/or datasets, among the available models on the TensorFlow Detection Zoo[9]. As this node is a implementation on a real time video streaming, measures can't be taken on *precision* terms. However, a deeper analysis of that kind can be performed with the JdeRobot tool `DetectionSuite` (described on section 1.3), which allows to measure performance in terms of advanced markers, as *Jaccard Index*, precision and recall.

The timing tests (Table 5.2) have been performed under the described available hardware, computing the mean inference time on a people walking video (Figure 5.11). Notice that the frame rate and/or resolution are not relevant, as every image experiments a reshaping before being feed-forwarded through the network.

[9] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

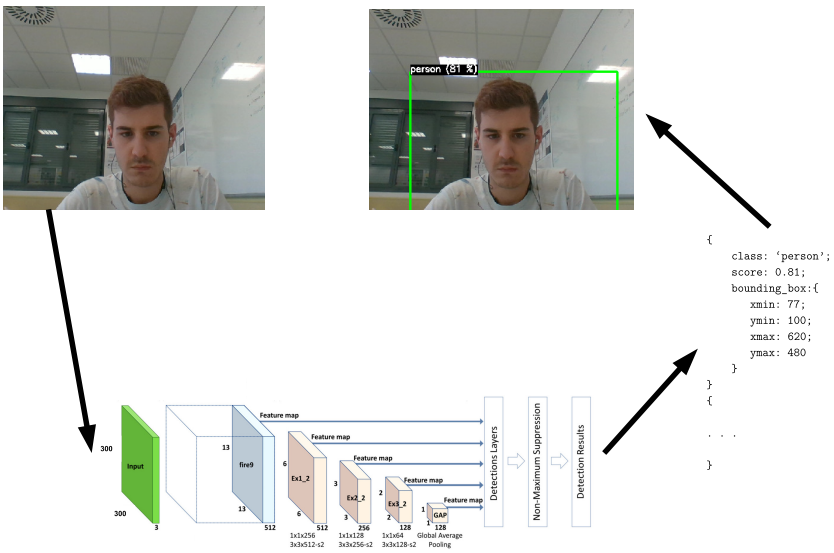


Figure 5.10: Information flow through the whole pipeline.

Architecture	Base network	Dataset	Mean inference time
a	b	c	d
a	b	c	d
a	b	c	d
a	b	c	d
a	b	c	d

Table 5.2: Timing performance tests for several modes.

Figure 5.11: Screenshot taken from the video used on the benchmark.

Chapter 6

`FollowPerson` node

6.1 Description

The previous node, `ObjectDetector`, shows such a fine real-time detection system, for several classes of objects (up to 90 in the COCO dataset case). It conforms an inert node, but that can act as a starting point for a ton of interesting applications to come up with.

In our project, we take one of these new derived applications, `FollowPerson` (Figure 6.1), and in the next pages we will study its insights. Our approach is a *reactive*[1] behavioral on a robot, which commands a Turtlebot robot to follow a target person (which we will call *mom*) using a RGBD[2] sensor. This person can be easily specified, providing the node an image of it, in the YML configuration file. The node will search and analyze its faces, and store it as the face to follow. It comprises, as we will see, some new systems as a modular extension of the pure detection component.

6.2 Node architecture

On the previous components, we had to support similar tasks, as the only radical change was the inner structure of the neural network. However, on this new component, we have to add a new element to the previous scheme, as we have a new task: *command movements to the wheels of the robot*. This has to be on an independent fashion, according to our design requirements (we can not implement blocking calls, so we need an asynchronous operation). So, the solution to make this possible is simply to implement *another thread*, which will be responsible of controlling the motors, based on the last network output. This way, we will have a schema similar to the one on Figure 6.2.

So, as a result, we can count up to 5 different threads:

- *Camera*: as we mentioned, we need the images from a RGBD sensor, which will be provided via ROS (`OpenNI2`), from the Asus Xtion sensor. These images will be delivered transparently through a ROS topic by the driver, and the Camera object will distribute them to the rest of threads.

[1] A reactive action has an immediate effect when the stimulus is perceived.

[2] *RGBD* stands for *RGB + Depth*. It provides, in addition to the standard RGB image, a depth map, as described at 3.1.2

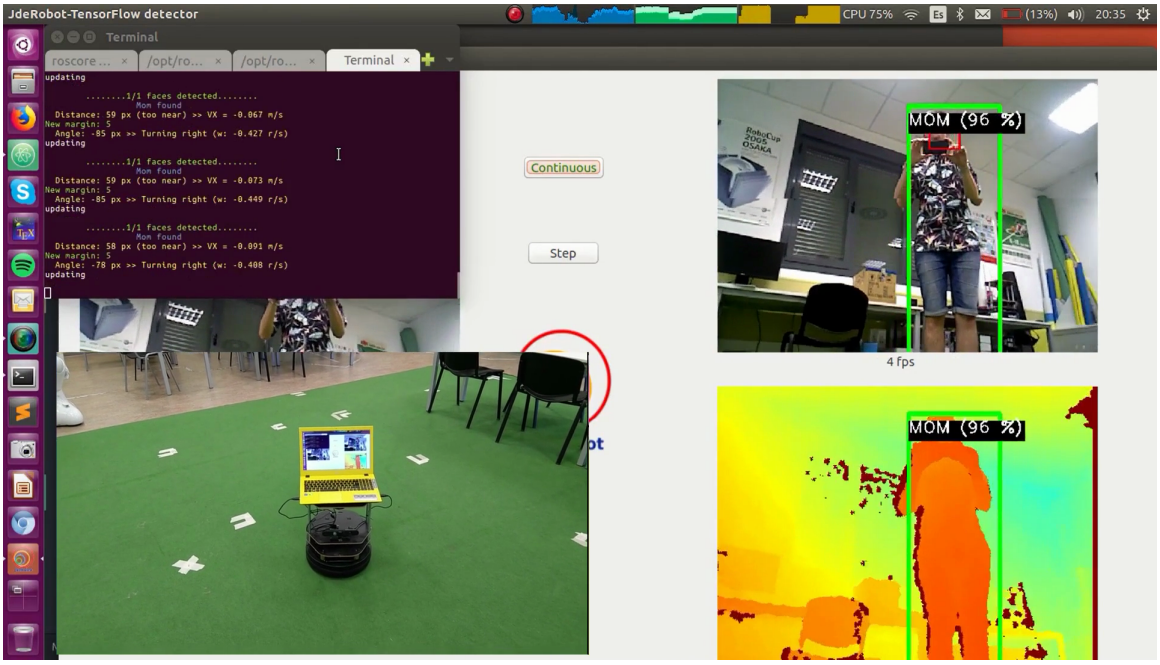


Figure 6.1: `FollowPerson` working (following mom).

- *Depth*: we obtain a depth map from the sensor, which is also delivered by the `OpenNI2` driver. One thing to mention here is that, originally, each pixel has a size of 16 bits, standing for the depth in millimeters detected on that particular direction. Unfortunately, through the transport process, it is converted to a 8 bits value, so we lose the real reference of the distance measure. To alleviate this, we will work with *relative* distances, comparing them with the range $[0, 255]$ they can take. This is 2^8 times less accurate, but it is fine given the application and the distance the sensor is capable to reach.

- *GUI*: this thread is, as before, destined to refresh the visible window, showing the incoming images (RGB image as in `ObjectDetector`, and depth image, tinted with an artificial *colormap* representing relative distances).

- *Network*: it behaves the exact same way than the `ObjectDetector`'s one, implementing a SSD CNN on a *MobileNet* base network, trained on COCO dataset. The tiny adaptations to this applications will be seen below.

- *Motors*: this is the new element. It is an asynchronous thread which analyzes the output of the network and, depending on the result of contrasting this information with both received images (RGB and Depth), sends a reactive command to the robot's motors. Even so, it keeps an intermediate tracker to soften the physical response, as it will be explained later.

6.3 SSD CNN Modifications

As said, the underlying CNN bundled on this component is the same SSD detector that `ObjectDetector` uses. However, a subtle modification of its output is performed before mak-

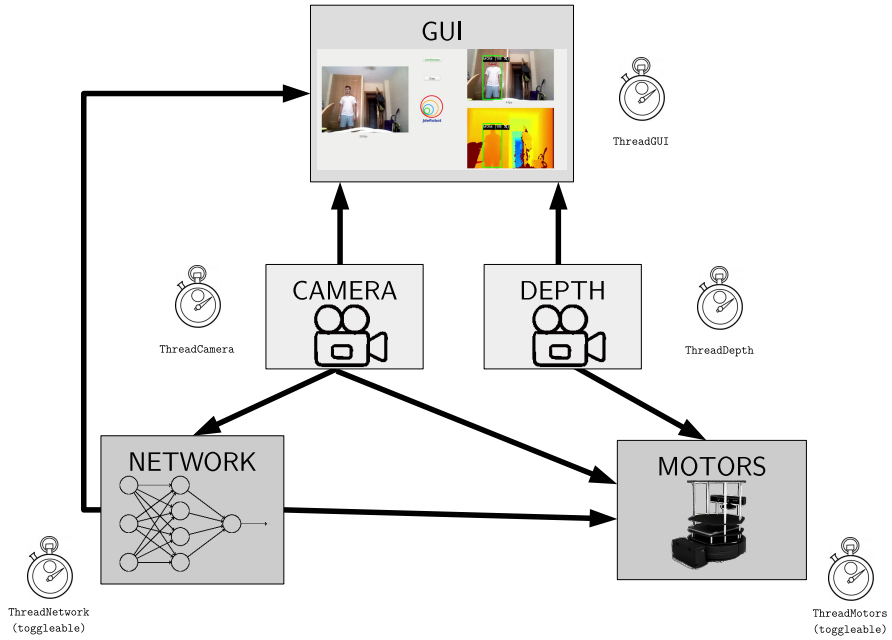


Figure 6.2: Architecture of the `FollowPerson` node.

ing it available to the GUI.

As our main objective in this node is to *follow a person*, we are not very interested in detecting other kind of objects in the image. In consequence, we perform a small filtering of the tensors returned by the output layer (`boxes`, `scores`, `predictions`):

```
...

(boxes, scores, detections, _) = self.sess.run([........])
# Now the last layer output is contained in those numpy tensors,
# We will work with the detection indices
# (they are identically sorted in the tensors).

# Firstly, we get the mask corresponding to the most confident
# predictions (to avoid false positives):
mask1 = scores > 0.5
# Then, we get the human detection indices:
mask2 = np.where(detections == 'person')

# Now we combine both masks:
mask = np.logical_and(mask1, mask2)

# And retrieve the most confident human detections,
# (their location and score).
correct_boxes = boxes[mask]
correct_scores = scores[mask]
```

```
...
```

Later, as it was done in the detection node, these filtered values are deposited in the accessible placeholder for `GUI` and `Motors` components.

6.4 Face detection and identification

The person following task can be already addressed, but our main interest is to be capable of tracking and following a single person.

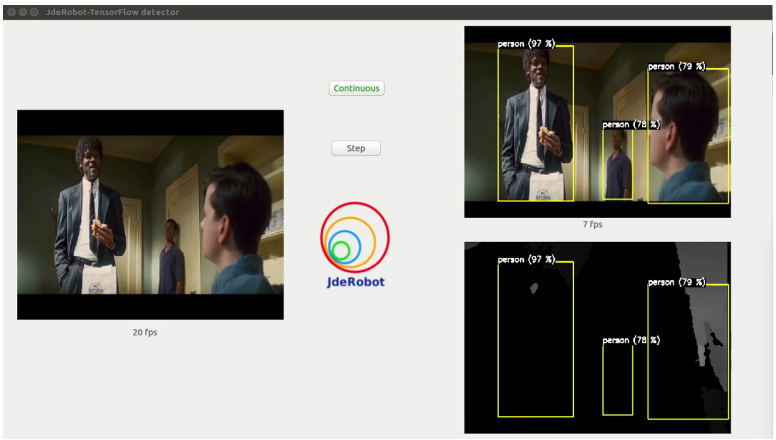


Figure 6.3: Who should we follow? Probably none of them.

We discern which one is the target person (*mom*), *identifying its face*. To do this, we have to firstly *detect* faces on the image, and then perform a *reidentification* with each found face, to contrast wether that person is or not mom. Now, we will describe the pipeline followed to perform these actions.

6.4.1 Detection: Haar Cascade Classifier

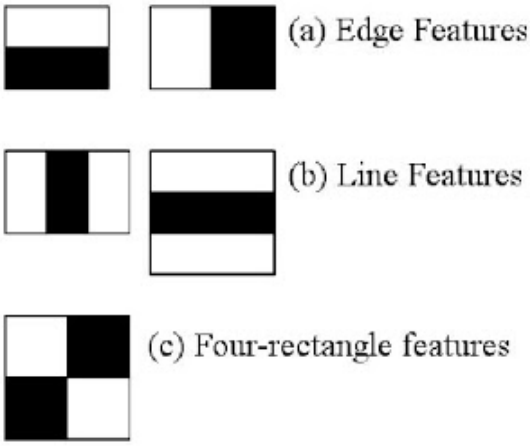
The face detection task has to be performed on the first place, and the chosen approach for this is the one described in [31]. This face detection algorithm, which is one of the most popular face detection Machine Learning techniques, comprises *Haar features*. The reason for this choice is the lower computational load[3] it supposes compared to other techniques, as it works with *grayscale images* (simpler than RGB ones), and the specification we have chosen is a simple algebraic computation with the pixel values, which result on a low-complexity system.

Haar features are binary *masks* (Figure 6.4a) that are slided through the *grayscaled* image. On each stride of a particular kernel/mask, the pixels contained on the black zone of the kernel are subtracted from those contained in the white zone (Figure 6.4b). The total result of these convolutions lets us know a promising zone to contain a face, or a clearly negative zone to discard. This takes benefit from the fact that practically on every situation, some determined

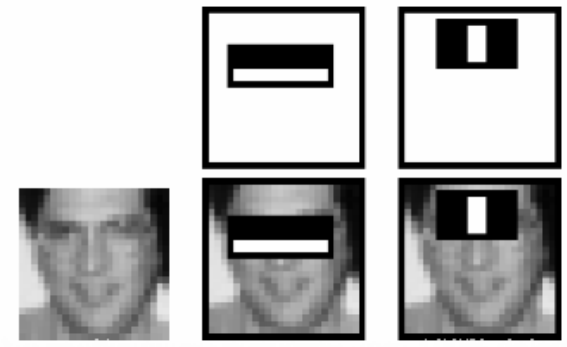
[3]We must remember that this will be running simultaneously with the SSD detector.

regions of a human face are darker than others. Hence, certain kernel dispositions will work much better on a determined zone of the face (Figure 6.4b).

As it can be figured, these *Haar* features are not trivial, they are obtained in a training, as a result of a ton of positive (and negative) examples analysis.



(a) Haar features.



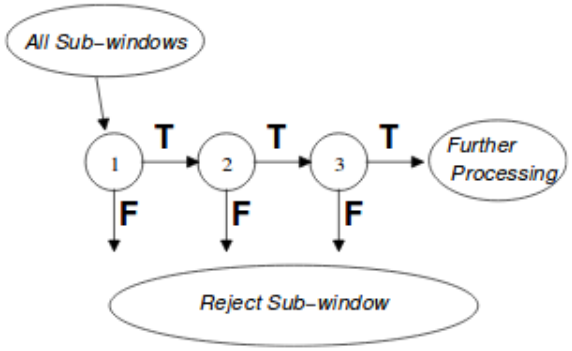
(b) Application of a Haar feature on a face image.

Figure 6.4: *Haar* features.

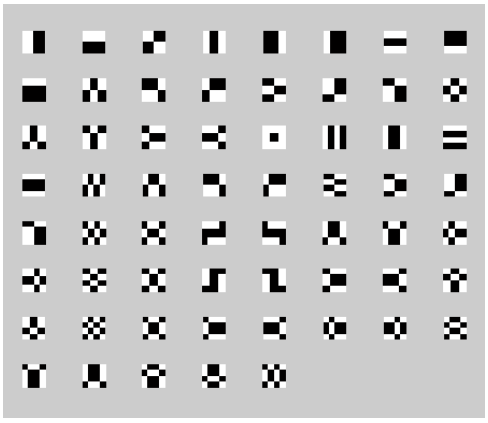
Once we know what a *Haar* feature is, we can describe the selected algorithm to perform the face detection: a *Haar-like feature Cascade Classifier*. This method, proposed on [32] by Viola and Jones, describes a pipeline to process an image, which consists on a *serie* of *Haar-based* checks.

The first stages check different areas of the image, using very simple *Haar* features, with the objective to discard regions (*sub-windows*) where certainly there is not a face. Those regions which pass the pixel subtraction test go forward to the next stage, where the process is repeated with a slightly more complex kernel (Figure 6.5a). On the final stages, where only a small region of the image could have passed all the previous *Haar* features, the most complex kernels (Figure 6.5b) are applied, discarding the possible false positives, and obtaining, finally, the regions where the system is sure of having detected a face.

Additionally, we can parameterize this detection process by the desired range of sizes to consider on a face, and the minimum number of neighbor features passed to consider a region as a candidate. We validated this system with good results in comparison to other classifiers (as LBP (*Local Binary Patters*) based), but this is the one which yields the best trade-off between satisfactory detections and a low computational load. This last factor is alleviated in some way, because we don't perform the face detection process on all the images, but *only in the currently detected persons*, hence it turns on a much lighter and efficient task that our real-time system can afford. Also, this allows us to keep a limp detection thresholds: if we obtain more than one face in a particular person, we will keep the highest one (as the face detection is constrained inside the person image). The handicap of this kind of classifiers is the difficulty to detect faces on a profile pose, as these features only apply to *frontal face* faces.



(a) Logical pipeline



(b) Increasing complexity of successive *Haar* features.

Figure 6.5: *Haar-like feature Cascade Classifier.*

The way to implement this system has been a OpenCV class, `cv2.CascadeClassifier`, fed with an XML containing the description of the stages (obtained on a training process and publicly available on OpenCV's official GitHub repository[4]).

6.4.2 Face Validation: FaceNet

If we have achieved to locate a person's face (it is not always possible since ambient conditions can be harsh sometimes), we can *reidentify it*. We will not deploy a identity storage system, or similar, as it is out of the scope of this node. Instead, we will use a *validation* system to check whether that particular person corresponds to the target to be followed.

We have implemented a solution based in [5], which develops a system called *FaceNet*. Its main functionality is to *map* face images to a 128-dimensional Euclidean space, where faces are represented by what is called *embeddings* (feature vectors on this specific space), and the distance between these embeddings directly represents *similarity between the faces*. This way, actions like face recognition, verification and clustering can become immediate, as they can be performed just over the obtained embeddings of each face. The mathematical definition of *distance* is the Euclidean generic one:

$$d(\vec{f_1}, \vec{f_2}) = \sqrt{\sum_{i=1}^{128} (f_{1_i} - f_{2_i})^2} \tag{6.1}$$

The advantage of this system in comparison with other approaches relies on the fact that it offers *strength*, as it runs a deep neural network underneath: the embeddings are optimized through a standard training process (using Stochastic Gradient Descent, with batches of tenths to hundreds of examples[5]), based on a particular cost function, called *triplet loss* (Figure 6.6).

[4] https://github.com/opencv/opencv/tree/master/data/haarcascades

[5] Again, we can notice that it is a simple process in comparison with non-deep-learning techniques, since we only need images of the faces labeled with their identity.

What is achieved is a minimum distance between the computed embeddings for images of the same face, and a maximum distance between faces of different individuals. This makes of it a really robust system, since these kind of networks, trained with images on different lighting and pose conditions, offer an excellent performance on real environments. In addition, it is a simple system, as the resultant embeddings are just 128-bytes values (easier to handle than other state-of-the-art system which implement PCA analysis, SVM classifications, etc.). Some other approaches, as *siamese networks* consist on a real-time comparison between two faces. This would make less sense in this scenario, as we will compare the current face with a *static* one, so it would be a waste of efficiency to compute the same values all the time for the reference face (*mom's*).

Figure 6.6: Triplet loss training. It minimizes the distance between an *anchor* (current example) and a *positive*, both of which have the same identity, and maximizes the distance between the *anchor* and a *negative* of a different identity (from [5]).

To sum up, we can appreciate that this is another perfectly suitable scenario for deep-learning, where it can perform much better in efficiency and simplicity[6] than other approaches.

The objective implementation[7] on this node is achieved, as with previous detection networks, with the *generic TensorFlow model loading*. We recover the graph and weights stored in the `.pb` model pretrained on the source repository (on the footer of this page), represented on Figure 6.7. As it can be seen, its structure is such a simple one.

Additionally, to ensure a robust response under different environment conditions, we include a preprocessing phase for every image entering to the network:

1. *Reshape*: the imported network works with 160×160 face images, so we have to resize the face to that particular shape.

2. *Blur* (with a small kernel): this is applied due to the probable low resolution and high noise that the analyzed face can suffer. It has to be kept in mind that the camera will be situated near the floor, looking upwards. This will cause the face to have a far position with respect to the sensor.

3. *Prewhiten*: with the objective of being insensible to light conditions of the image, we perform a normalization process of the face, eliminating the light information. This is achieved with a standard normalization operation on each color channel, being x the matrix of that channel of the image, μ its mean and σ its standard deviation:

$$x' = \frac{x - \mu}{\sigma} \tag{6.2}$$

[6]Perfectly compliant with *KISS* principle: https://en.wikipedia.org/wiki/KISS_principle
[7]Inspired on: https://github.com/davidsandberg/facenet

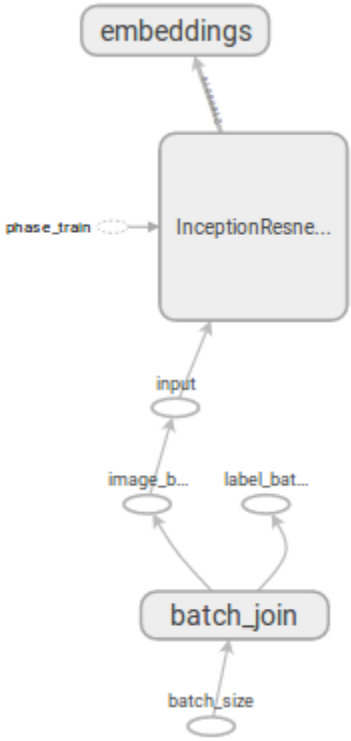


Figure 6.7: *FaceNet* architecture.

So, we obtain a normalized image, which is centered on 0 and scaled on a same way (by its standard deviation). We achieve much more comparable images with this method.

The final result of this preprocessing pipeline can be observed at Figure 6.8, where we obtain the normalized faces, ready to be processed by the *FaceNet*, which will compute the embeddings. We can compute the resulting L^2 Euclidean distance (Equation 6.1), through the methods we have implemented on the `FaceNet` class:

```
...
img1 = imread('img1.jpg')
img2 = imread('img2.jpg')

# We discard the second output (it contains the face bounding box).
face1, _ = facenet.getFace(img1)
face2, _ = facenet.getFace(img2)
# The next method performs the face preprocessing, feed-forward pass of
# both faces, and compute the distance.
distance = facenet.compareFaces(face1, face2)
...
```

The last thing to do is establish a *threshold* distance, to decide whether the two compared faces are the same or not. The experimental tests drove us to the same values that the paper [5] specifies: a distance threshold of 1.1 (notice that at Figure 6.8) the three faces have a respective distance between them below that threshold, because they belong to the same person, even at different lighting conditions as it can be appreciated.

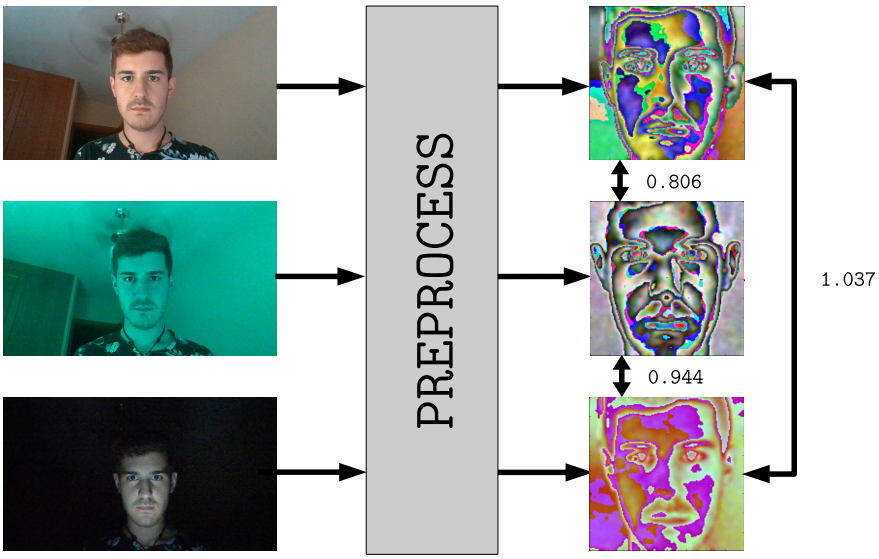


Figure 6.8: Preprocessing result on several conditions (the different colors in the output images are due to the color mapping performed by the plotting backend), and L^2 distances computed between the faces.

As we mentioned before, we will use this small deep neural network to compare all found faces with *mom*'s (which will be a constant, as it comes from the reference image provided to the node). So, first of all, we compute *once* *mom*'s embeddings, and compare the live obtained embeddings from new found faces with *mom*'s, just computing the distance between them. As this is a methodical task, it has been included as another method inside the class: `compareWithMom(face)`.

6.5 Face and Person Trackers

So far, we are able to detect the seen persons in the image, and the faces inside these persons. After completing these two process successfully, we are ready to compare a certain face with *mom*'s. However, although the person detection CNN is extremely robust (which is its main advantage), it can miss a detection sometimes, because of the lighting or confusing environment around the person. On the other hand, the face detection algorithm (*Haar-like feature Cascade Classifier*) is more susceptible to failure, as it is a simple (although efficient) method, which is not designed for difficult situations. This is as alleviated as possible inside the detection pipeline[8], but *false positives* are feasible even so.

For this reason, we have inserted an *intermediate* block between the detection algorithms (person, face), and the face validation: the *trackers*. These trackers are a security measure to

[8]As we mentioned before, we have implemented a very limp detection conditions, filtering the detected faces and keeping the highest one.

avoid false positives and false negatives. As its operation is the same in both cases (persons and faces), we will describe it on the generic case, working with *instances*, and concretely with its bounding boxes (which determine its position).

The tracker divides the instances into three sets:

- *Detected instances*: the fresh instances which have just been detected in the image (`DetectionNetwork` or `CascadeClassifier`).
- *Candidate instances*: the detected instances which are during the process of being a tracked one.
- *Tracked instances*: the candidate instances which have passed a period of *patience*, and are currently being tracked. These are those we will consider to look for *mom* and/or compute a physical response. If we lose these instances we will keep thinking for a marginal lapse of time that it is still there.

This way, we are not trusting directly the raw output from the detectors, but trusting instead the combination of that output with the past detections. We will avoid *false positives*, as *detections* will have to be for a while on the *candidate* set, and we will also false negatives, as *tracked* instances will stay for a while in that set even if they are not detected, before being untracked. So, *our movements will be dictated by the tracked stimuli, not by the detected ones*. This will eliminate noisy responses and get a much softer behavioral than trusting directly the imperfect output from a classifier.

Its operation and exchanges between sets is the same in both cases (person and face), using a counter (owned by each instance) that is varies between 0 and the patience value (we have set 5 as a prudent number):

1. When a new instance is detected, its position goes into the *detected* set.
2. Its position is compared[9] with each *tracked* instances. If it is suspiciously near of a *tracked* instance, it is taken as the new detection for that instance. So, the *tracked* instance position is updated (taking the new value), and its counter is incremented in 2 units. The *detected* instance is removed of the *detected* set.
3. If it does not belong to any *tracked* instance, we try the same for the *candidate* instances. If we are successful with one of them, this *detected* instance is taken as the new position for that *candidate* one, so its position is updated and its counter is incremented in 2 units. The *detected* instance is removed of the *detected* set.
4. If we were not successful in any of the previous cases, this instance is a completely new detection, so it is appended to the *candidate* set (with a counter value of 1), where it will stay for a while if it keeps being detected.

This described procedure is repeated with every new detection. When it has finished, before ending the iteration, a *general update* is performed over all the *candidate* and *tracked* instances:

[9] All the distance comparations between instances are made computing the module of the distance vector (in px) between their centers.

1. The counter of all the *candidate* instances is checked. If it has reached the patience value, it is considered that it has been seen for too much time for being a false positive, so it goes into the *tracked* instances (with a counter equal to the patience + 1).

2. If any of the *candidate* instances has reached 0 on its counter, that means that it has not been detected anymore, so it was a false positive (it was not detected long enough to move to the *tracked set*). It is removed from the *candidate* set.

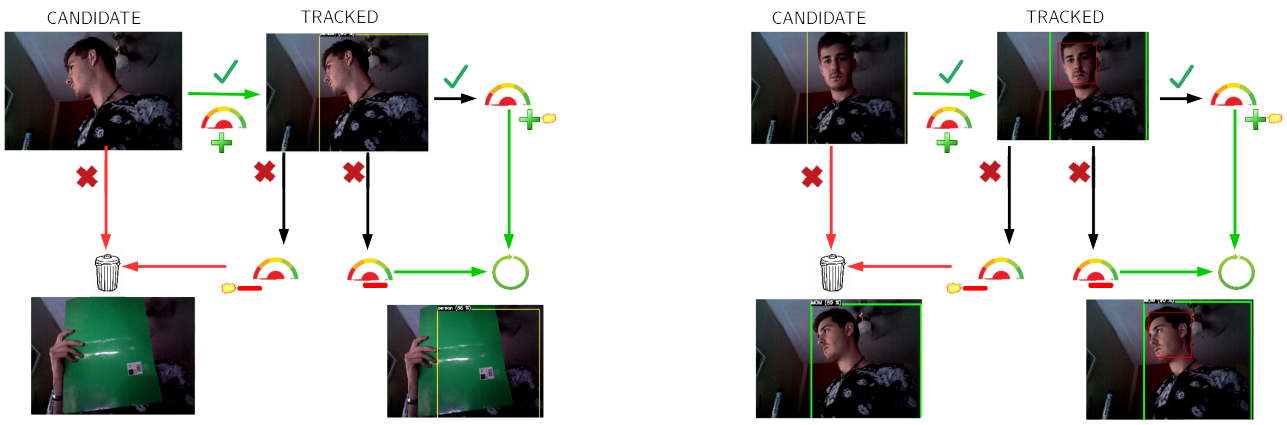
3. The same step for the *tracked* instances: if any of them has reached 0 on its counter, it is removed (it was a *tracked* instance which has been lost for enough time for not being considered a false negative).

4. The counter for all the *tracked* and *candidate* instances is decremented in 1.

When a strong detection is mantained, it is stored in the *tracked* set, and its counter is always saturated to the maximum value, so it is safe for being demoted for a while if it stops being detected for a few frames. This can be applied to person detections (avoiding undesired detections) and to face detections (this cancels the effect of the limp face detection, where only the highest one will be kept and, in addition, it will have to be detected for a few frames to be considered).

This role promotion and demotion system is successfully implemented with custom types in Python, where we have a general `PersonTracker`, which keeps the coherence between the sets for all the image, and a `FaceTracker` inside each person (to have three of the explained sets being executed for each detected person) yielding excellent results to keep steady detections. This allows to handle faces only inside detected persons (which also eliminates false positive face detections outside a person, which is never convenient).

The general effect of the inertia provided to the confident detections allows to have a much softer response in case of fleeting false positives/negatives. We can have a glance on this behavioral on Figure 6.9.



(a) Tracking process for a person.

(b) Tracking process for a face.

Figure 6.9: Schema followed by the trackers.

6.6 Physical response

The main objective of this node is to *command movements to the Turtlebot motors*. We need to know how to behave, and which will be the good movements to command to arrive to *mom*'s position.

6.6.1 Follow algorithm

Once we have detected, tracked and identified *mom*, we have to keep a certain intelligence, or behavioral schema, to know what to do on each possible scenario. The *flow chart* of the implemented behavioral is represented on Figure 6.10, where we can see that *it is enough for us to identify mom once*. Given that it will be tracked, we can automatically update its position, without the need to see its face again to confirm that it is mom indeed. If we are following mom without seeing its face, and we find its face on another detected person, the role of *mom* will swap to this new person, who will be followed from that moment.

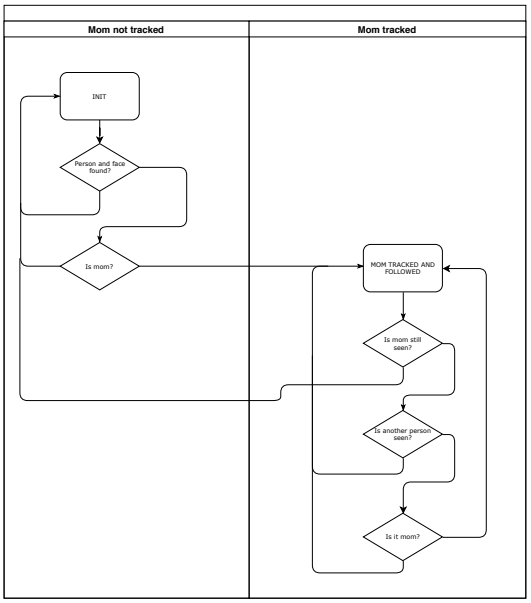


Figure 6.10: Following behavioral (flow chart).

In last place, if we don't see *mom* our robot stays in its place, slowly turning towards last *mom* tracked position, in order to find it again.

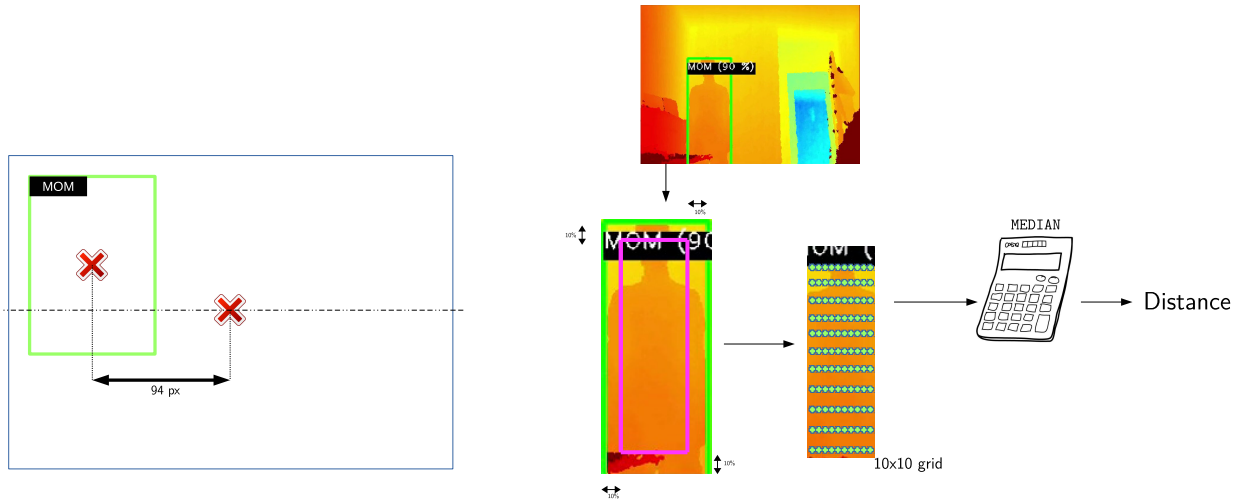
6.6.2 Position calculation

The fact of knowing, thank to all the previous blocks, where *mom* is inside the image lets us know where (relatively to the robot) it is, given that the RGBD sensor is aligned with the Turtlebot frontal face. This means that, what is seen along the center of the image will be just in front of the robot, on a straight line. Thus, we will be able to act in consequence on both of the motors interfaces that the Turtlebot offers:

- *Angular*: the first thing we have to figure out is if the robot has to rotate towards *mom*. This can be computed with the bare *mom*'s bounding box. If we compute its center, we can know the horizontal distance from it to the entire image center. This will give us the *horizontal error* (measured in pixels, as it is a relative amount) (Figure 6.11a).

- *Distance*: the other error to compute is the distance one. This one is figured out on a slightly different way. As explained at subsection 3.1.2, a *registration* process carried out by the driver maps the depth pixels into the RGB ones. So, if we know *mom*'s position in the RGB image (its bounding box), *its depth measures will be right inside the same box in the depth image*. That is the source of the measure. The implemented approach

to obtain a single value (as we need it to send the proper command to the motors) is to perform a *spatial depth sampling* inside the depth slice, cropped with the bounding box. To be cautious, we keep the depth image inside a safety margin of 10% of the image (to be sure we sample inside the person). Then, we create a 10×10 *grid* inside this new crop, and retrieve the depth values measured on those points (a total of 100 measures). The total estimated distance will be the median[10] of that set of measures (Figure 6.11b).



(a) Horizontal error estimation.

(b) Computation of the median distance to *mom*.

Figure 6.11: Computations of both errors.

6.6.3 PID controller

Once we know the relative *mom*'s relative position from the robot, we have to send the proper commands to the robot, to make it follow *mom*. On the first place, we have to know when to act and when to stop the motors. As we don't want the robot to go to the exact place where *mom* is, we have to establish a *safe/dead zones*, where we the robot won't move further towards *mom*. These zones are represented in Figure 6.12.

As our desirable relative position is to be aligned (the center of the bounding box in the center of the image) and at a safe distance (at ± 10 px from the reference distance), we will rotate and/or move in a straight line in consequence, trying to carry *mom* inside the safe zones. Hence, we have to *compute a proper response* to move the robot towards the correct direction (independent responses on the angular and linear dimensions). Instead of computing just a response on a proportional way to the current error, we can implement a *closed loop feedback system*, which keeps in mind the previous readings and responses computed, to get

[10]We choose the median for being generally more robust than the mean, as it is not affected by the value of outlier measures (an accidental sample on a background surface would alter the mean on a significant way).

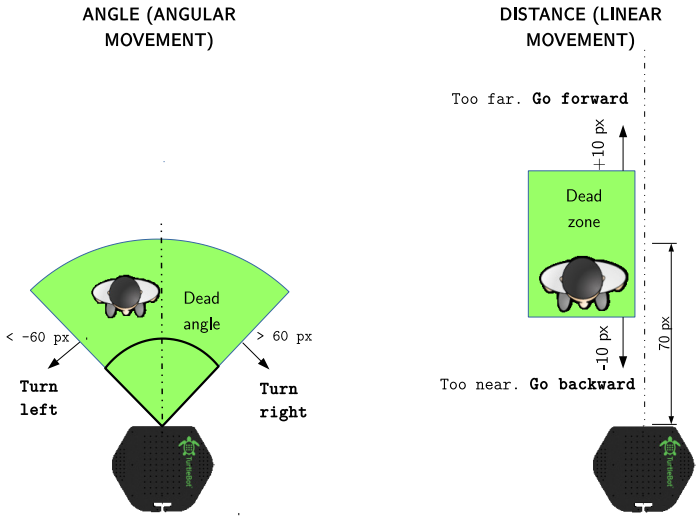


Figure 6.12: Safe zones. The robot will consider *mom* as correctly followed inside them (on a separate way for each dimension).

a better fitting to the ideal response in that moment[11]. Concretely, we will implement a *PID* controller [33], which is the most common form of feedback at industrial applications (more than 95% of the control loops are of this kind). PID stands for *Proportional, Integral and Derivative* control, and computes a total response ($u(t)$) following the next formula (modified for discrete time, as we are on a digital system):

$$u[n] = k_p e[n] \quad + \quad k_i \sum_{i=0}^{n} e[i] \quad + \quad k_d(e[n] - e[n-1]) \tag{6.3}$$

For those who did not stop reading, we will analyze this formula, which combines three sub-responses:

- *Proportional*: $k_p e[n]$. This is the basic component, that computes a response directly proportional to the measured error.
- *Integral*: $k_i \sum_{i=0}^{n} e[i]$. Here we compute an additional response, equivalent to the summation of the total error until now. This way, although a proportional response is not enough and the error gets stabilized in a non-zero value, the system will accumulate that error, getting *annoyed* with the time. So, we will get a bigger response the bigger the total error is[12].
- *Derivative*: $k_d(e[n] - e[n-1])$. This part stands for the *difference* between the last measured error and the current one, and it quantifies how is the system responding[13].

[11] Otherwise, we can suffer oscillations and peaks on the response, as this application is a system affected by noise and possible errors in the measure. That is the reason why we implemented all the past softening blocks.

[12] When the monitored variable goes into the tolerated zone again, the total error has to be reseted, as it won't be necessary for now.

[13] On systems without inertia, this contribution is generally ignored, having a simple PI control loop instead.

If the difference has a high value, that means that the system is on a far state/position with respect to the last iteration. So, in order to eliminate the *inertia* the system could have acquired (which might bring oscillations and overshooting), the derivative part acts, braking or accelerating the command depending on the observed response to the previous one.

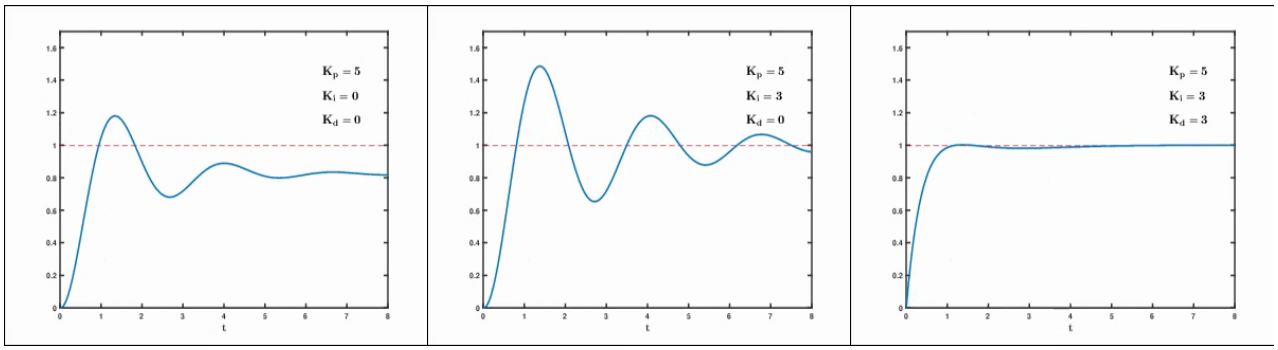
As we can see, the combination of the three responses can achieve a much faster and steadier response (Figure 6.13), bringing back the system under control on a fast and efficient way. Each contribution is parameterized by its corresponding constant (k_c, k_i, k_d), so a critical task is to find the optimum value for each one of them.

For our implementation, as we have implemented two independent PID controllers (for linear and angular speeds, respectively), we have experimentally looked for the most suitable values for each one of the parameters, obtaining the combination on Table 6.1. It is important to mention that these are not the final values that regulate the movement (as the angular and linear errors move in different scales, a difference of 1 pixel in the angular error means almost nothing, but much more in the linear one). For this reason, and to keep reasonable values, they are scaled by internal constants inside of each PID controller.

	Linear	**Angular**
k_p	2	7
k_d	0.1	0.5
k_i	3	10

Table 6.1: Optimal found values for the parameters in each PID controller.

As the rule that governs the controller is the same (Equation 6.3), we have created a generic class (`PIDDriver`), which controls a motor with a PID closed loop, adding extra functionalities as soft reactive responses (to avoid abrupt movements), and limiters in the response, to mantain control over the robot.



(a) Proportional. (b) PI. (c) Full PID.

Figure 6.13: Different controllers response along time.

This way, we compute a suitable response with a PID controller for each dimension, and the mentioned class sends it directly to the motors (thank to the possibility of passing the

move function as an argument to the generic controller).

So finally, we have the final code schema of the functionality added in top of *ObjectDetector*:

```
...
# jdrc is the comm communicator.

# We create the client to move the robot:
motors_publisher = jdrc.getMotorsClient('FollowPerson.Motors')
w_function = motors_publisher.motors.sendW
v_function = motors_publisher.motors.sendVX

# Controllers creation:
w_PID = PIDController(w_function,
                      Kp=_,
                      Ki=_,
                      Kd=_,
                      scaling_factor=_,
                      limiter=_)

w_PID = PIDController(w_function,
                      Kp=_,
                      Ki=_,
                      Kd=_,
                      scaling_factor=_,
                      limiter=_)
...

# Measure of the errors:
angular_error = [subtract centers]
linear_error = [sample depth of person]

# Computation of the response (PID controllers):
w = w_PID.computeResponse(angular_error)
v = v_PID.computeResponse(linear_error)

# The controllers automatically send the response on a
# reactive but soft way.
```

At the end of the each iteration, we get the robot moved towards *mom* (the target person to follow), making use of all the previously defined blocks (Figure 6.14), as it can be seen on the final result, posted on the project Wiki[14].

[14] http://jderobot.org/Naxvm-tfg

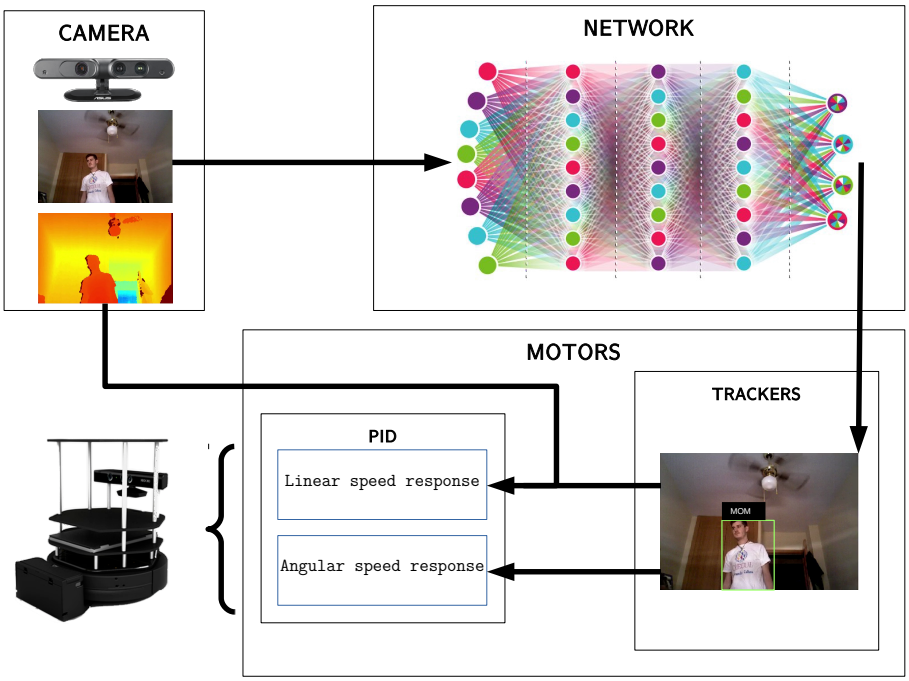


Figure 6.14: Functional diagram of the `FollowPerson` node.

6.7 Experiment: PTZ Camera

As an alternative experiment or approach, we can implement this tracking and following system on a PTZ camera. As explained in subsection 3.1.1, these cameras are supported by servo motors on vertical and horizontal axis, that allow the camera to move as a *mechanical neck*.

We can consider an alternative system, which implements the previous pipeline (with slight modifications), using this camera (Sony EVI D100P, on Figure 3.1) as the *actuator device*. Thank to the Python distribution easiness using packages, this case can be implemented as an additional package, being able to modify as less as possible. So, `FollowPerson` commutes pertinent functions between the Turtlebot and the PTZ packages in runtime, just selecting the used device in the YML file.

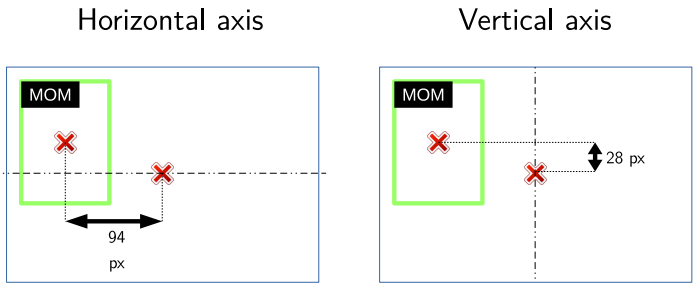
The movement commands are now destinated to pure *rotations* of the neck, so the movement update on each axis (vertical/horizontal) has to be computed on a similar way than the horizontal case on the Turtlebot (Figure 6.11a). So, our approach in this case follows the computation on Figure 6.15a

However, as the movement commands are stored in a queue buffer, we are limited to perform *differential* pose updates (e.g. move 1 or 2 degrees on each axis towards the objective per iteration), as described in Figure 3.2. Hence, a PID controller does not make sense here, as the set of feasible outputs is discrete and non adjustable:

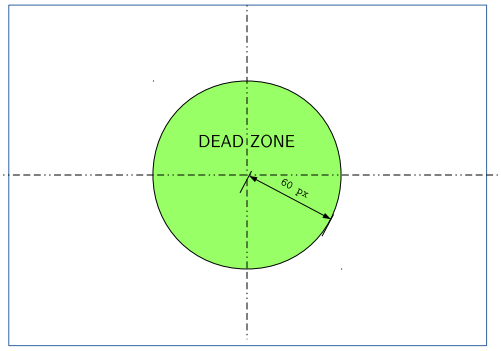
- 0 if the error stays inside the dead zone (Figure 6.15b).
- ± 1 if the system lost *mom* (in order to slowly move looking for it).

- ± 2 if mom is being tracked. The camera performs an increment of that magnitude towards mom in each axis.

Finally, we can take advantage of the most of the standard `FollowPerson` total algorithm, obtaining the schematic algorithm on Figure 6.16.



(a) Error computation on each axis.



(b) Dead zone of actuation (radius of 60 px).

Figure 6.15: Error computation parameters on the PTZ case.

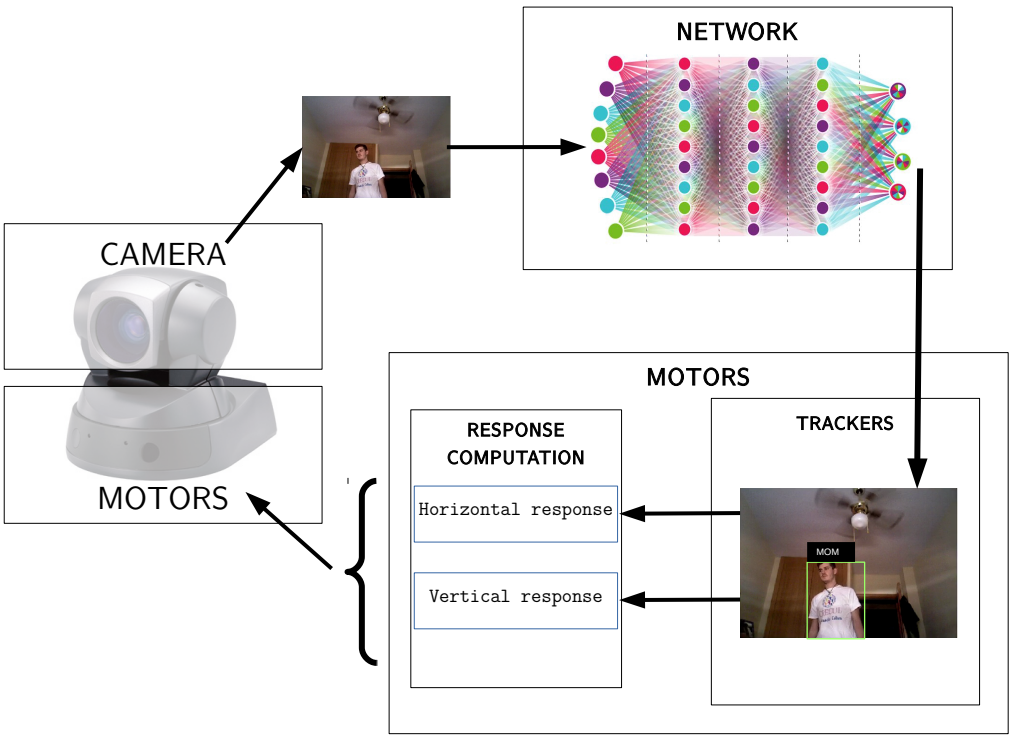


Figure 6.16: Total schema followed in the PTZ case.

Chapter 7

Conclusions

7.1 Conclusions

This final chapter will be destined to revisit the initially proposed objectives (chapter 2). With a complete coverage of the performed tasks, as it has been described on the previous pages, we can contrast what has been achieved on each one of these objectives.

Classification

The first objective required to upgrade the scope for an existing JdeRobot component, `DigitClassifier`, designed to perform *classification* tasks using *deep learning* techniques.

Its functionality was emulated using the *deep learning* framework TensorFlow (core of this project), obtaining satisfactory results on the real-time digit classification task, training an *in-house* neural network on our own, and obtaining initial knowledge about the framework and enough deep learning skills to move forward to more complex tasks.

In light of the excellent achieved performance, the brand new TensorFlow implementation and the previous one (made with the Keras framework) were merged into an *official JdeRobot* component[1], capable of commuting between both frameworks.

Detection

After having accomplished a basic domain on *deep learning* with *classification* tasks, we tackled a more ambitious milestone: *detecting objects on a real-time operation* (where there was no previous reference in the JdeRobot environment).

Although the process of training a detection network was ruled out of the scope of the project (because of computational issues, as stated previously), we addressed the detection task using publicly available *pretrained networks*. So, we developed a wrapping

[1] https://github.com/JdeRobot/dl-digitclassifier

TensorFlow environment to *abstract* the model (architecture, dataset on which it was trained, output format, etc.), and moved the neural network processing to a *GPU environment* (to achieve the optimum refresh rate for a real-time operation).

This remarkably efficient detection framework was demonstrated to be capable of real-time processing, so we have developed an entire node, `ObjectDetector`, to visually perform this task on an *incoming image stream* (abstracting the source).

Hence, the final result has been a node displaying the bare current image, and aside the same one with the detected objects overlaid, indicating its location making use of a *bounding box*, in addition to the decided *class* the object belongs, and the *score* (standing for the reliability level that prediction has).

Once more, the excellent performance on real-time (using detectors with a SSD architecture under the shield) has driven us to integrate this node[2] in the JdeRobot environment as well, developing another package to do the same in Keras. This has enabled us again to be able to abstract the frameworks, and toggle one of them through the YML file. This component can be taken as a passage to *alternative implementations* making use of the robust detection performance that *deep learning* can yield (and not only drawing the results over the image). In addition, this has given us the capability of *benchmarking* new models, as they can be transparently loaded into the created environment.

Tracking and following

The two previous milestones allowed us to accomplish *state-of-the-art* purposes in the *Computer Vision* field. As it has just been stated, `ObjectDetector` is a powerful tool to take advantage of a real-time detection system.

So, as the final research objective, we have considered an *actuation* system, which uses a powerful *neural network* to accomplish an extra objective. In order to overlap our research with the prosperous field of *robotics*, we have developed a component capable of *following* a specific person (*mom*). This has been achieved ringing into our set of tools concepts like a *PID* controller, or behavioral schemes, which have throw in some extra knowledge.

Additionally, this has demonstrated the possibility of finding *synergies* between *deep learning* and another kind of scientific fields, as it is currently made in tasks like processing sound, words, medical images, advertising, etc.

Personal fields

The previously described infrastructure we have created has allowed a novice investigator to check how important a *correct coordination* is, when it is necessary dealing with tasks

[2] https://github.com/JdeRobot/dl-objectdetector

like a *conjoint* development (as it had to be done at certain moments).

In addition, a telecommunication engineer must know how to multiplex resources, even itself. The closest approach a human can achieve to this is *TDMA*, so it's a key factor to be able to *make a correct time assignment over incoming tasks*. This has allowed to learn the importance of keeping work tidiness and rigor, as well as a suitable level of *compromise* (sustained through motivation to attainment), in order to be adaptive and be ready for a change of plans, or a readjustment of the requirements. All of these capacities make the difference between a long-term research project, and a simple homework task.

On the other hand, in this kind of project it is important to keep a decent trace of the made work. This is essential in order to be able to go back in any moment, to revert some undesired effect, or just for consulting a past detail which we need to bring back. For this purpose, it has been very convenient to maintain the *project's Wiki page*, for being able to have a glance on the achieved advances, and keep a trace of the temporal progress of the whole project.

In the professional point of view, it has been essential to acquire further skills about *version control systems*, as Git. If we take a look back, carrying through all the progress we made would have been no other thing than a hell. This can be an important benefit, as every development project in a corporate environment makes use of this kind of controls.

7.2 Future research lines

The proposed milestones on this project have been successfully achieved using a useful and innovative tool as *deep learning*. Hence, it opens some interesting doors to future researches or improvements:

- *Upgrade DigitClassifier:* for now, this component looks for the digit in a fixed window inside the image (the central square). Another module could be implemented to perform a *character detection* in the image, using *deep learning* as it is a really efficient implementation.

- *Translate the nodes to a compiled language:* one of the main handicaps of Python is the fact that it is an interpreted language (much slower than a compiled one). So, a *translation* to a lower level language as `C++` would be very interesting, as it is a widely supported language in this framework, and there are already very efficient *deep learning* implementations on it (as *Darknet/YOLO*).

- *Use a deep learning face detector in FollowPerson:* the main advantage of the *deep learning* systems is the robustness on the operation, so a facial detection system implemented with this technology could be a powerful resource to be able of performing a facial detection/validation in harshly lightened environments.

- *Multimodal person detection/tracking:* some extra functionality could be squeezed of a RGBD sensor, like *tracking a person in complete darkness*. As *deep learning* systems offer good results distinguishing a person silhouette, we can perform people detection on a depth image (in fact, that is just what is being made in Figure 1.9).

- *Add a navigation algorithm to FollowPerson:* the only movement commands sent to the Turtlebot are controlled by the relative position of the person with the robot. However, as the robot incorporates a laser sensor, we can add an *obstacle avoidance* system, in order to perform a non-blind navigation towards the person.

Bibliography

[1] machinelearningguru.org. Image Filtering - Machine Learning Guru. http://machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1.html. 6, 15

[2] Alexis Cook. Global average pooling layers for object localization. https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/. 6, 15

[3] ros.org. Writing a Simple Publisher and Subscriber (Python). http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29. 6, 29

[4] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015. 7, 54, 56

[5] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015. 7, 65, 66, 67

[6] J. Potel. Trial by fire: Teleoperated robot targets chernobyl. *IEEE Computer Graphics and Applications*, 1998. 9

[7] P. Berkelman and Ji Ma. The university of hawaii teleoperated robotic surgery system. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2565–2566, Oct 2007. 9

[8] A. M. Okamura. Methods for haptic feedback in teleoperated robot-assisted surgery. *Ind Rob*, 31(6):499–508, Dec 2004. 16429611[pmid]. 9

[9] Daniela Girimonte and Dario Izzo. *Artificial Intelligence for Space Applications*, pages 235–253. Springer London, London, 2007. 9

[10] Bengio Y. LeCun Y. and Hinton G. Deep learning. *Nature*, (521):436–442, 2015. 11

[11] Eric Roberts. Neural networks - history. https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html. 12

[12] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. 12

[13] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, 1949. 12

[14] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947 EP –, Jun 2000. 13

[15] D. Pascual. Study of convolutional neural networks using Keras framework. 16, 32, 36, 40, 49

[16] N. Oyaga. Análisis de aprendizaje profundo con la plataforma caffe. 16, 32, 40, 49

[17] M. Pieras. Visual people tracking with deep learning detection and feature tracking. 16

[18] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, August 1986. 21

[19] ProfessionalQA.com. What is Spiral Model in Software Development Life Cycle? http://www.professionalqa.com/spiral-model. 21

[20] R. Calvo. Comportamiento sigue persona con visión direccional. 27, 31

[21] python.org. What is Python? executive summary. https://www.python.org/doc/essays/blurb/. 27

[22] ros.org. What is ROS? wiki.ros.org/ROS/Introduction. 28

[23] ros.org. openni2_launch (official ROS package page). http://wiki.ros.org/openni2_launch. 30

[24] José M. Cañas. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. 30, 31

[25] Irwin Sobel. An isotropic 3x3 image gradient operator. 02 2014. 43

[26] S Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015. 47

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 EP –, Oct 1986. 48

[28] readthedocs.io. Loss Functions: Cross Entropy. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy. 48

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 48

[30] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. 55

[31] C. Awadallah. Nuevas prácticas docentes de robótica en el entorno jderobot-academy. 63

[32] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–511–I–518 vol.1, 2001. 64

[33] Karl Johan Åström and Richard M. Murray. Feedback systems: An introduction for scientists and engineers. Technical report, 2004. 73