uc3m | Universidad Carlos III de Madrid

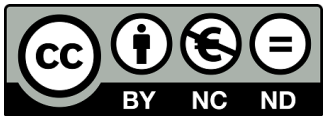
Master Degree in Telecommunication Engineering
Academic Year (e.g. 2019-2020)

Master Thesis

"Embedded solution for person identification and tracking with a robot"

Ignacio Condés Menchén

Fernando Díaz de María
Eduardo Perdices García
Leganés, 2020



SUMMARY

This project describes the development process of an embedded system capable of performing a reactive following of a person. It makes use of convolutional neural networks and probabilistic tracking for processing the perception acquired by a RGB-D camera. This input is processed in a NVIDIA Jetson TX2, an embedded System-on-Module (SoM). This device is capable of performing computationally demanding tasks onboard, coping with the complexity required to run a robust tracking and following algorithm. The full design is implemented on a robotic mobile base, which receives velocity commands from the board, intended to move towards the desired person.

Keywords: deep learning, robotics, person following, rgbd

DEDICATION

yes

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This chapter presents the motivation that led to the development of the proposed work, as well as a review of the state of the art, to achieve a general panorama of the problems and methods that this work addresses. Later, the general objectives of the developed system are outlined, with a summary of the structure of the work.

1.1. Motivation

Last decades, the production prices of digital cameras and high-resolution sensors have been greatly reduced, bringing these devices into the consumer market segment: nowadays, everybody carries at least 2 cameras in their mobile phone, aside of high-quality web cameras, or even driving-assistance cameras in cars. This, beside an increase in the hardware performance has resulted in a strong drive for the computer vision research (Figure 1.1): there are many possibilities out of industrial environments for applications using cameras, such as fancy image modifications, or autonomous driving, as it can be seen on Figure 1.2.

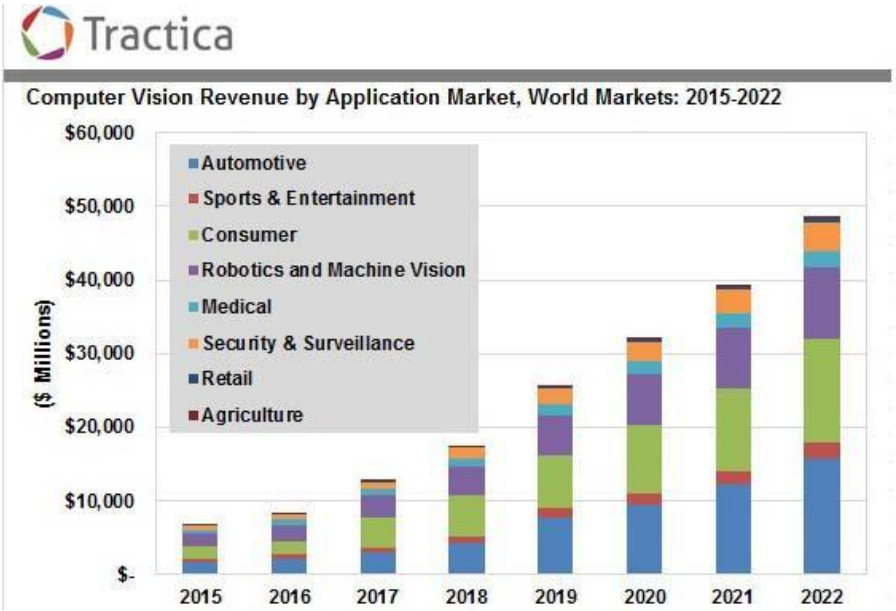


Fig. 1.1. Computer Vision revenues in the last year, and forecast for 2022 (source: [1]).

Specially, the latest times have been notoriously active in this field because of the massive use of *deep learning* for addressing high complexity tasks, such as language understanding [2], speech recognition [3] and computer vision problems, which are linked to the growing interest shown in Figure 1.1. This massive use began in the ImageNet classification contest, where a deep neural network system, AlexNet [4], achieved an overwhelming victory over another approaches. This discovery, along with the strong

(a) Modifications of a subject on a portrait, such as apparent gender, or age.

(b) Autonomous driving on a Tesla Model X.

Fig. 1.2. Examples of contemporary computer-vision applications.

advances in computing power and parallel computing have impulsed the usage of these systems, which show an outstanding performance with the available means nowadays [5].

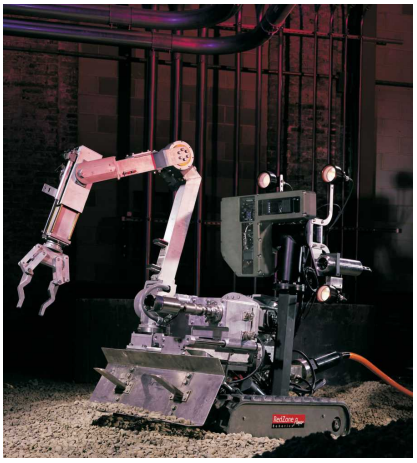
On the other hand, robotics applications can be really useful at daily tasks. These tasks are of greater interest when the behavior of a robot tends to emulate the human one[1], with the advantage of no people exposed to a significant risk, or, in a less gloomy scenario, without human body physical limitations.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

Teleoperated robots: this kind of robots are capable of perform certain actions, which are *remotely controlled by a human operator*. This type is the mostly used one on the hazardousness (Figure 1.3a) [6] or precision [7]. Some advances are made nowadays improving the teleoperation function, implementing *feedback* from the robot, such as haptic feedback [8], or VR (*Virtual Reality*) sensation, to allow that person to sense the environment as if they were in the robot position.

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 there are some factors (as the time elapsed performing an action, or the cost of a control link with the robot) 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 behavior* (Figure 1.3b), so some actions can begin to be performed autonomously with a certain intelligence, as it will be describe below.

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

[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 reacting to external stimuli intelligently.

Fig. 1.3. Example of a teleoperated and an autonomous robot.

There are outstanding synergies between robotics and computer vision, as it is explored on the system proposed in this work: these fields are combined for obtaining a robust robot capable of following a certain person, navigating towards it on a reactive behavior, and using deep-learning based visual perception. This behavior is composed of two main components: the *perception block*, in charge of processing the images from an RGBD camera placed on the system, and the *actuation block*, which moves the robotic base accordingly to the relative position of the person to be followed.

This application can be specially interesting on social robots, which are designed to follow a person at home or in a hospital. According to [10]: *robots that operate around people in the real world need to move in coherent, easily-understood ways, so that they will not startle or harm the people around them. In particular, for robots that operate in hospitals or in nursing homes*.

The work proposed in this thesis improves the system developed on [11], where a neural following system was run in a standard laptop, with a camera and a robot plugged. In the following dissertation, this work will be revisited, and the points of interest which have allowed to enhance the previous version will be described.

The main contributions of this solution may be brought in as follows:

Embedded solution: the final system is mounted on a battery-powered *mobile base*. This robot features a high-performance GPU embedded on a SoM (*System-on-Module*). In contrast to the previous work, this assembly can operate on its own,

without requiring an external computer to perform the deep learning inferences or running algorithms in parallel. A remote monitoring of the behavior is available as well, but it is not required for the system to work. In addition, specific optimization engines allow the system to run faster with 3 neural networks than previously with only 2 networks, on a low-consumption hardware. This will be reviewed in chapter 3.

Person identification: the proposed system runs 3 neural networks. These networks perform inferences over the images perceived by the RGB-D sensor, which is attached to the system as the sensing source of the robot. The inferences are devoted to detect the different persons in the scene, as well as to distinguish them by means of a discriminant feature: their face. Unlike the previous development, all the detection and identification tasks are based on neural networks, achieving greater robustness and reliability as it will be depicted in chapter 3.

Tracking: the full system includes also a person tracker based on optical flow. This aims to guess the trajectories followed by each person that the robot can see. As opposed to the previous work, this tracker allows to roughly follow the persons while the neural network yields a new update, as this tracker takes considerably less time to infer the person displacement. As a result, the robustness of the entire system is improved, compared to a version governed exclusively by the neural inferences, which are sensitive to visual occlusions as well. Trusting just on these inferences could easily result on an unsteady behavior. However, the introduction of the tracker softens the robot movements ensuring a more robust following, as it will be explained on section 2.3.

1.2. State of the art

As it was previously introduced, this work is performed to explore the synergies on robotics and deep-learning-based visual perception. In this section, the current approaches and tools will be depicted in order to outline a general panorama where this work may be placed.

The problem to be addressed is to *get a robot with a camera to follow a person*. This problem can be split into several steps, where different approaches have been previously proposed. These steps will be covered in the following subsections.

1.2.1. Visual person detection

One of the most used approaches is commonly called the *Viola-Jones* detector [12]. This algorithm relies on a *rigid body model*, which fits a specific shape. On a grayscale image, this shape can be typically distinguished by means of the pixel intensity levels. Several

spatial filters called *Haar features* (Figure 1.4) are introduced: these are used across the image looking for the intensity pattern for each mask, which should resemble a part of the rigid body. As this presents a weak decision by itself, several filters (previously chosen in a training process) are combined on a *boosted cascade* (Figure 1.5). A person is detected if the weighted combination of several filters are triggered inside a certain area, which is decided to potentially contain a person [13].

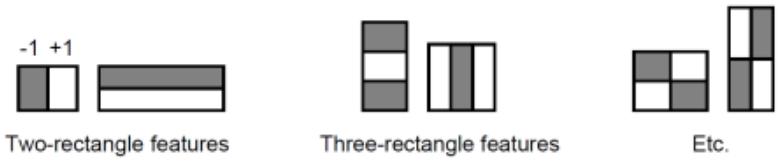


Fig. 1.4. Haar features: some examples [13].

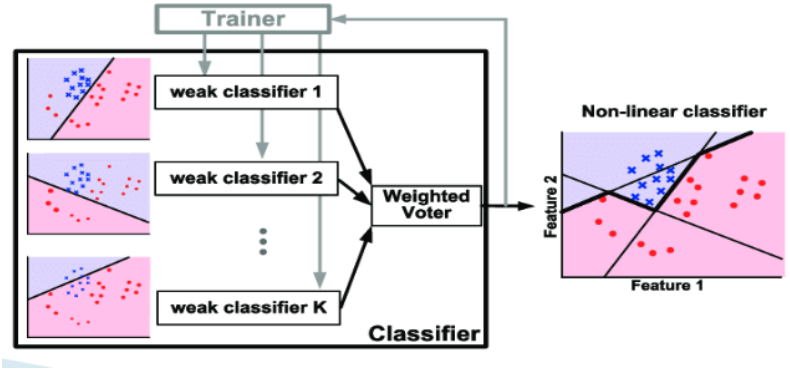


Fig. 1.5. Boosted weak classifiers [13].

Although this system was originally developed to detect faces, the rigid body model allows a generalization powerful enough to extend this to another object classes, *person* among these. The open-source standard image processing library, OpenCV, includes pre-trained models[2], which can be directly used with their Viola-Jones implementation. Scale invariance can be achieved evaluating the image at multiple scales on runtime.

Another common approach nowadays for person detection is based on HoG (*Histograms of Gradients*) [14]. This method computes local features by means of the intensity gradients across the image, and quantizes them using their angle (creating a histogram for the oriented gradients for that pixel), as it can be seen on Figure 1.6.

These gradients are collected in 64×128 windows, and treated as features from which a linear SVM (*Support Vector Machine*) is trained in order to classify a region as *person/non-person*. Figure 1.7 shows the average gradient patch for a person (the direction of each gradient is not shown). A visual inspection immediately resembles the shape of a person standing up. Thus, this detector will yield the best performance when the person to be detected stands in that specific pose.

[2] https://github.com/opencv/opencv/blob/master/data/haarcascades

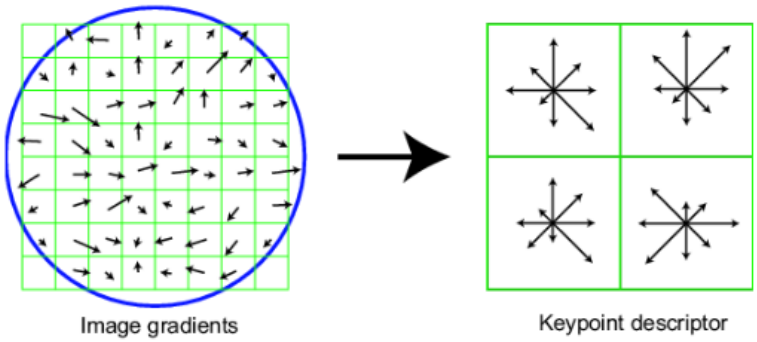


Fig. 1.6. Example of a HoG, quantized to 8 directions [15].

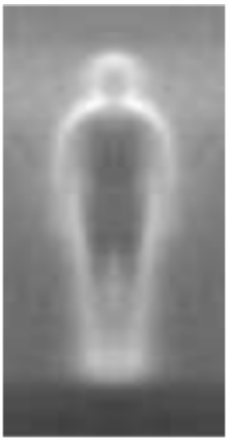
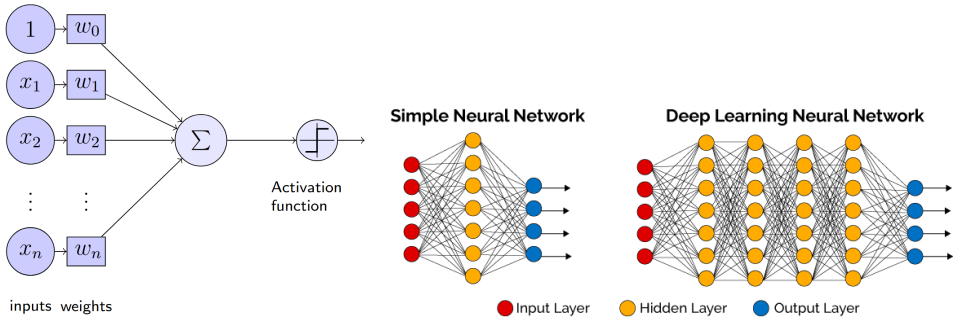


Fig. 1.7. Average gradient for person detection on [14].

These methods, among several more, have been the state-of-the-art techniques: the cornerstone are the image gradients, which can be computed with a high efficiency, and present decent performance. However, their main drawback is the *generalization* capability, as a successful detection is highly dependent on the person pose. However, in the latest advances, the detection techniques have moved towards the spreading paradigm: *deep learning*, especially the most salient tools on image processing: CNNs (*convolutional neural networks*).

CNNs are based on standard neural networks, which combine lots of neurons or *perceptrons* organizing them into layers. These perceptrons (Figure 1.8a) implement non-linear operations, that allow to extract (after a proper training process) abstract features, which gain in complexity when the number of internal layers increase. When a neural network is composed by several *hidden* layers (in addition to the input/output ones), it is placed into the *deep learning* paradigm (Figure 1.8b).



(a) Basic unit of a neural network: the perceptron (source: [11]).

(b) Standard neural network vs. deep neural network (source: [11]).

Fig. 1.8. Basis of deep neural networks.

Based on this approach, and taking advantage of the *spatial correlation* when the signal to process is an image, a neural network can be modified to implement a different operation on each perceptron: the *image convolution* (Figure 1.9).

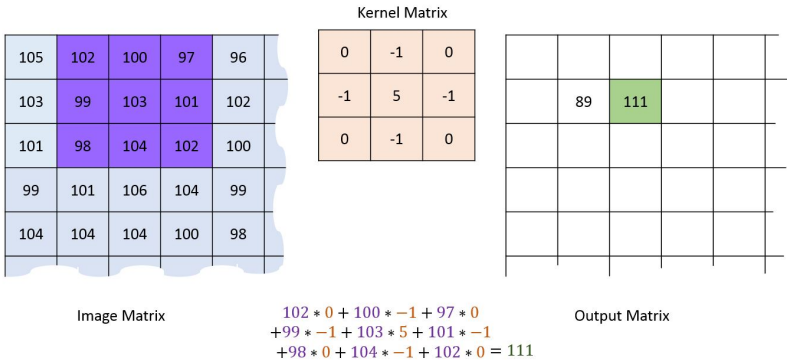


Fig. 1.9. Convolution applied on an image, applying the mask (red) on a region (purple) of the input image, storing the result on the mapping of the central pixel of the region (green). The computation is the sum weighted by the mask values (bottom) (source: [11]).

As it can be seen on Figure 1.10, convolutional units may be arranged conforming a set of layers used to build *feature extraction* stages (shown in red in the figure). Several layers can be concatenated, gaining in depth and obtaining more complex feature maps. These layers are finally followed by a detection/classification ensemble of *dense* layers (shown in blue in the figure): a set of layers with standard perceptrons fully connected among them, yielding a final output, dependent on the classification structure of the network.

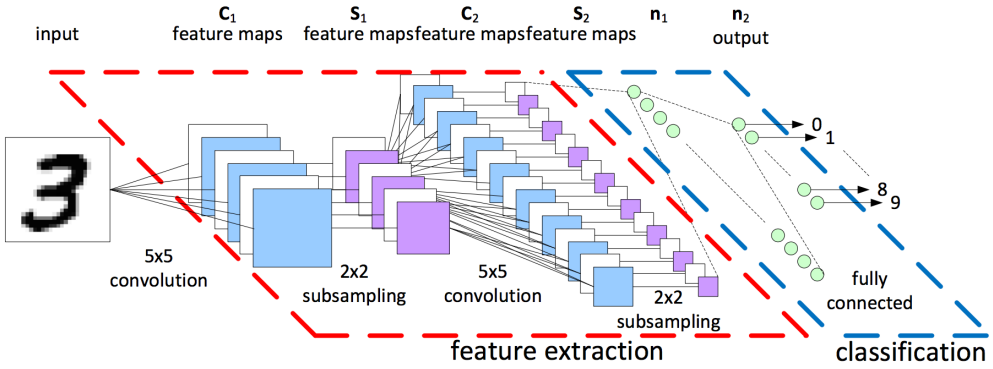


Fig. 1.10. Schematic of a digit classification CNN (source: [11]).

In the case of object detection networks (the ones involved in this work), the output varies depending on the implementation, but it is generally composed of a set of (location, probability) tuples, one for each class the network is capable of detecting. Figure 1.11 shows the activation maps of an object detection network, where the map presents higher values in the regions with high probability of containing the object of the class it is designed for. Several of these maps compose each neuron on a CNN.

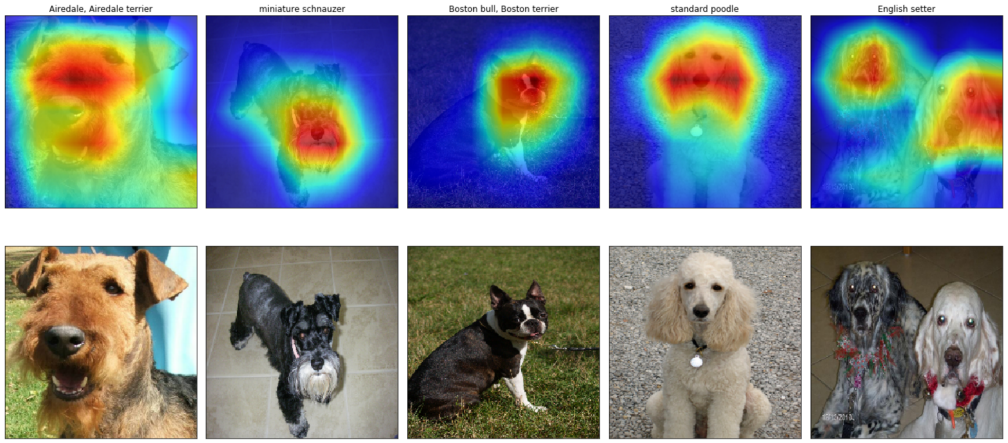


Fig. 1.11. Activation maps of a detection CNN searching for dogs on different images (source: [11]).

One possible application of this concept is focused on what is called *Region-based Convolutional Neural Networks* [16], which require a previous step on the image called *region proposal*. This step is devoted to find potential regions on the image to contain an object. This way, the challenge is to label these region according to the objects contained inside, reducing the problem to a classification task. However, the process to find these undetermined regions and iterate over them makes the process too slow for real-time requirements, which are explicitly contained in our requirements. Care has been put in posterior works [17] [18] to reduce this computation time. However, this reduction in time brings about a reduction in precision as well.

SSD

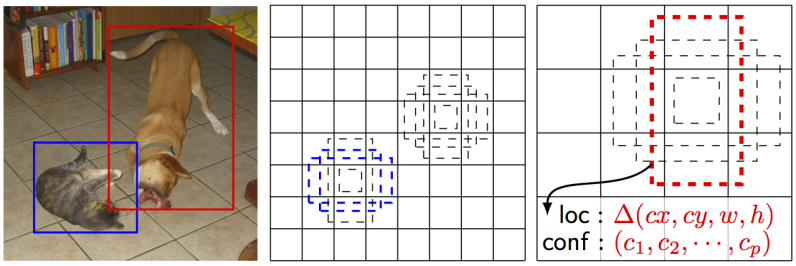
Another outstanding object detection architecture is SSD (*Single-Shot Multibox Detector*) [19]. The main benefit from this architecture is the fact that it embeds all the required computations in a single neural network, reducing the complexity compared to other approaches requiring external region proposals, as it was depicted above. This greatly reduces the computational time when the network has to evaluate an image. The architecture can be seen at Figure 1.17, and can be split into stages [11]:

Reshape: the posterior stages evaluate the images on a fixed tensor size of $n \times 300 \times 300 \times 3$ (being n the size of the input batch). Other image sizes might be used, however this one offers a good trade-off between score and computational load.

Base network: this first group of layers are reused from a typical image classification model, such as VGG-16 [20]. The first layers from this architecture are utilized in this design, truncated before the first classification layer. This way, the network can leverage the *feature maps* from the classification network, in order to find objects inside the input image. At the output of this network, several convolutional layers are appended, decreasing in size. This has the objective of predict detections at multiple scales. One thing to mention at this point is that the base network can be a different one rather than VGG-16, such as a MobileNet [21], which is highly optimized for running on low end devices. This is interesting as our embedded system will be limited in computing power. It will be revisited in future sections.

Box predictors: for each extracted set a dedicated operation is performed, generating a small set (typically 3 or 4) of fixed-size *anchors*, with varying aspect ratios for each cell on a grid over the activation map (Figure 1.12). As these maps have different sizes, this aims to detect objects in different scales. The anchors are then convolved with small filters (one per depth channel), which output *softmaxed* confidence values for each known class, and offsets for the generated bounding box. So, for each detected object (on that scale), the network computes the score for each class and its estimated position inside the feature map (hence, in the image as well).

Postprocessor: as several detections might be triggered in the same area for different classes and scales, a *Non-Maximum-Supression* [22] operation is performed at the output of the network to retain the best boxes, under a combined criteria of detection score and IoU score (*Intersection over Union*), which measures the overlapping quality between two bounding boxes, as it can be seen in Figure 1.13.



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

Fig. 1.12. A set of boxes are generated centered on each point of every feature map [19].

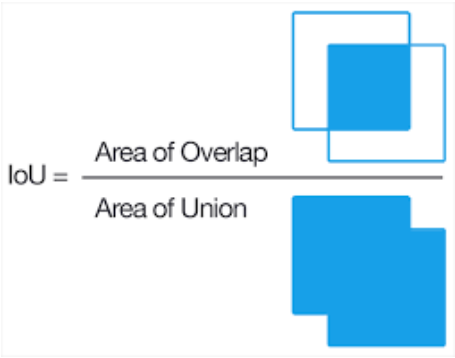


Fig. 1.13. Graphical representation of the IoU score between two bounding boxes.

YOLO (You Only Look Once)

Another interesting approach on the neural networks field is the YOLO (*You Only Look Once*) system [23]. Its main advantage is its inference speed, due to the fact that it performs a single analysis on the entire image, dividing it into a grid of cells. Each cell predicts up to 5 boxes, containing an *objectness score* (the predicted IoU of the proposal with an object, regardless its class), the coordinates of the bounding box, and a probability for the object belonging to each class. This design run faster than other methods [23], however it presents a poor performance when detecting small objects.

This design was revisited in YOLO9000 [24], introducing several improvements such as batch normalization on the input of the convolutional layers, or the concept of *anchor boxes*: the box proposals follow a fixed set of aspect ratios, chosen previously using clustering on a training set. As it can be seen on Figure 1.14, limiting the proposal shapes to 5 fixed sizes improves the performance while maintaining a high IoU metric. A visual inspection shows that the selected anchors seem like a reasonable shape for the majority of the objects the network aims to detect. Additionally, the number of deep layers was increased from 26 layers to 30, and a semantic modeling is performed on the labels across different datasets, allowing the network to be trained in different datasets under a common semantic structure called *WordTree* (Figure 1.15).

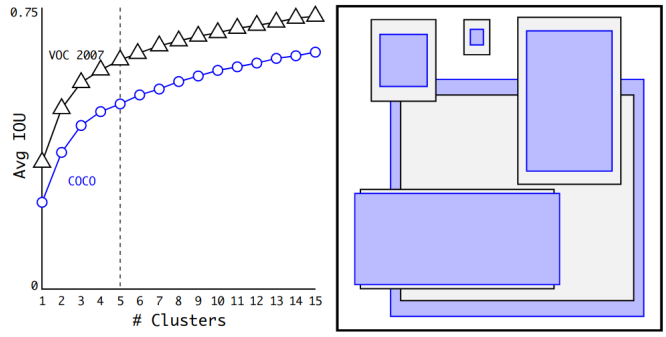


Fig. 1.14. Result of the anchor k-means clustering on VOC and COCO for YOLO9000. Using $k = 5$ anchor sizes on the right yields a good tradeoff between simplicity and improvement on the obtained IoU with respect to using $k - 1$ clusters (source: [24]).

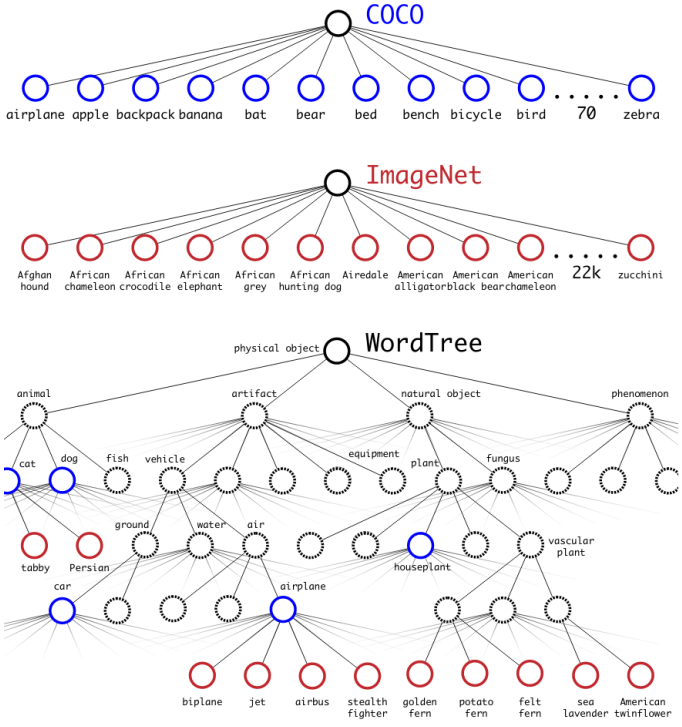


Fig. 1.15. Comparison between simple labeling structures (top) and a WordTree semantic grouping under categories. This allows to follow a dataset-agnostic training process as the labels can be combined using WordTree.

The latest improvement of YOLO, YOLOv3 [25], features residual networks [26], which tackle the problem of *vanishing gradients* when the networks become deeper. The stacking of several layers results on gradients diminishing its value up to a point the precision mode of the machine is not able to handle. The gradients are canceled, burdening the training process, as the first layers parameters take a substantially higher time to converge. The residual networks added in this revision of the design add shortcut connections across the layers, centering the backpropagation gradients on 1. As this reference states [25], the combination of these residual layers and convolutional ones

allows to train much deeper architectures (53 convolutional layers), capable of yielding a higher generalization. As in the SSD detectors, the YOLO architecture performs multi-scale detections, using 3 scales for splitting the feature maps into cell grids. A similar k-means than in Figure 1.14 is performed on the COCO dataset, selecting 9 anchor sizes instead of 5, and grouping them in 3 scales. Now, on each of the cells, 9 anchor bounding boxes are fit (3 anchor shapes $\times$ 3 scales). This aims to solve the poor performance of the previous version when dealing with small objects, as well as a better generalization: in the R-CNN [16] and the SSD [19] the anchor shapes are hand-picked. These changes, with a tuning on the error function, conform the YOLOv3 improvements over the previous versions.

For each (anchor, cell, scale) combination, this network predicts:

- The coordinates of the object inside the anchor. Details can be visualized on Figure 1.16.
- *objectness* score, which is computed by means of a logistic regression in order to determine the probability of overlap with a ground truth bounding box more than any other prior anchor.
- 80 scores, as the original implementation is trained in the COCO dataset, which contains 80 classes. These classes might be overlapping (e.g. “woman" and “person"). Thus, these scores are computed by independent logistic classifiers and are not passed through a *softmax* operation.

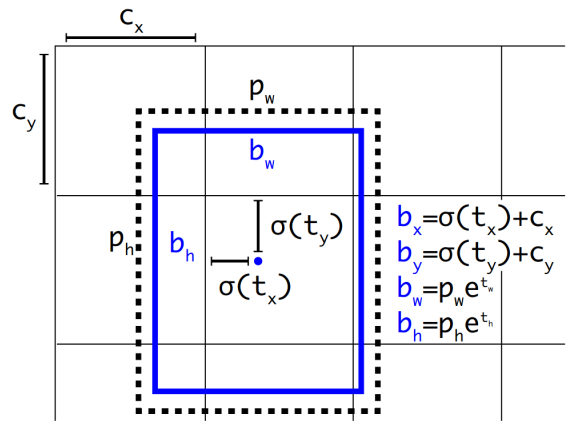


Fig. 1.16. Output on YOLO for each anchor and cell. The dashed line represents the prior anchor, while the blue line represents the detection which corrects that anchor.

The architecture of a YOLO-based detection network can be seen beside a SSD-based one in Figure 1.17. This allows to see the fundamental difference in the feature extraction stage of each approach.

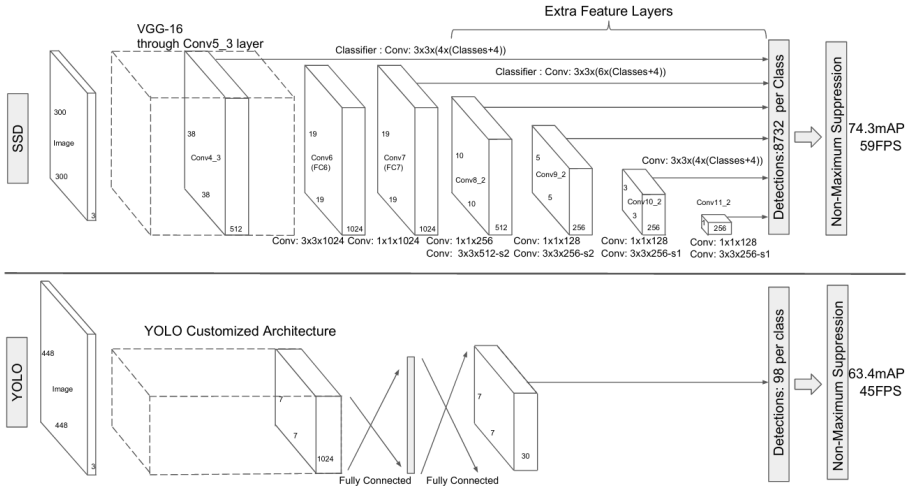


Fig. 1.17. General architecture of a SSD network (top) and a YOLO one (bottom).

1.2.2. Person identification

On a controlled environment, where the only present person is the one to be followed, a person detection system could be enough for following purposes. However, in a regular scenario, there might be several people inside the field of vision of the robot. This problem can be approached by means of a distinguishing feature of the person of interest, provided beforehand. One example is [27], which computes the color distribution of the person of interest, and later compares this distribution with the ones belonging to the different persons using the Bhattacharyya coefficient [28] (a measurement of similarity between two probability distributions). This metric can be applied to measuring the similarity between the color histograms of the reference person and the detected one. However, this system can be deceived replicating the color distribution of the person of interest: wearing similar clothes helps to reduce the distance between the histogram, leaving a chance to confound another person with the one to follow.

A more robust approach is to use the *face* of the person as the discriminant feature, as its uniqueness makes it a good reference to identify the detected person. As it is summarized in [29], several applications extract facial *landmarks* from the morphology of a given face (Figure 1.18), and use them to classify the face, comparing it against a set of known faces and estimating the identity based on the distance to each known face. Some open-source libraries such as `dlib` and `OpenCV` provide the algorithms to perform these processes.

The intuition behind these methods are to *project* the image of the face into a lower dimensionality space, which allows to extract significant features from each face. These features have to be consistent for the same face across different pose and lighting conditions (Figure 1.19). An useful transformation when a dimensionality reduction is

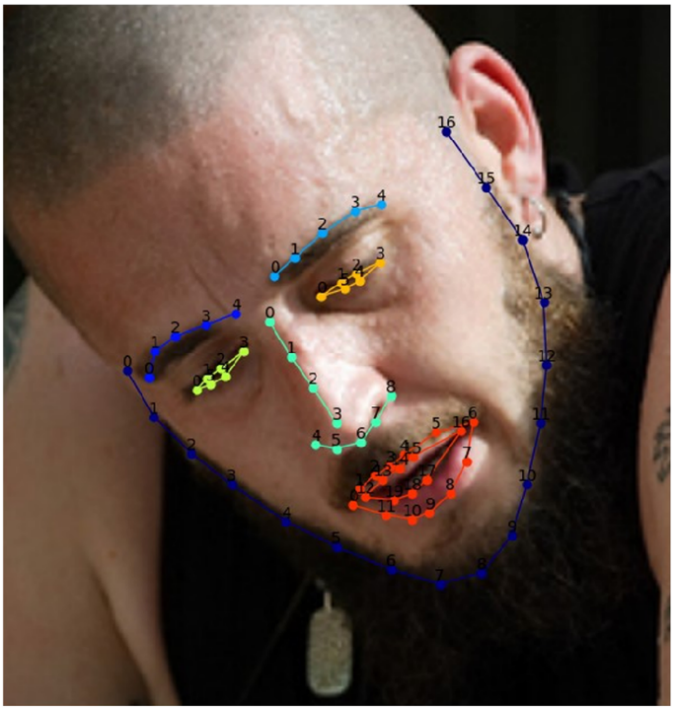


Fig. 1.18. Facial landmarks are dependent of the face shape and morphology (image from [29]).

pursued is PCA (*Principal Component Analysis*), a linear transformation that can be implemented to deal with the face recognition problem [30].

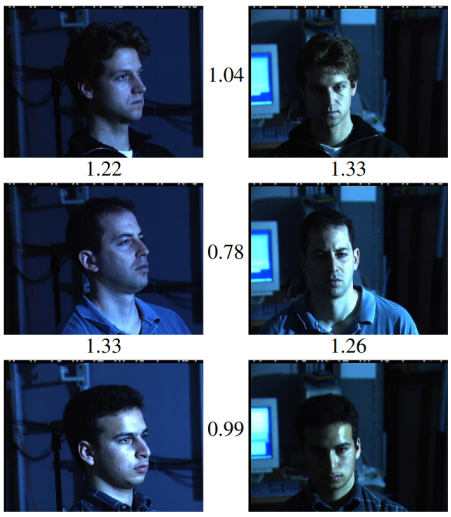


Fig. 1.19. Examples of poses and light conditions across which the face projections are desired to be consistent for the same person (image from [31]).

Deep learning face identification: FaceNet

However, once again neural networks can be leveraged in order to achieve this and more: as the PCA is a linear operation, it could be learned by a single layer neural network. Thus, the introduction of deep networks can yield interesting results. The most relevant approach so far uses deep convolutional networks for performing this process [31], implementing an architecture called *FaceNet*, which is partially based on the Inception [32] module, designed by Google researchers in order to greatly reduce the number of parameters in a neural network. What this network computes is called an *embedding*, a projection of the input face image into a point in a 128-dimensional hypersphere. This allows to translate the identification into linear algebra terms, such as *distance* between two faces, as well as clustering and applying unsupervised algorithms in order to determine the identity of a trivial face, among a collection of known regions. The architecture can be visualized in Figure 1.20. These networks can be trained using a loss function called *triplet loss*, inspired by the work in [33]. Given a training sample (*anchor*), a *positive* example (same class than the anchor) and a *negative* example (different class than the anchor) are chosen, and the network is tuned to maximize the *anchor-negative* embeddings distance, and minimize at the same time the *anchor-positive* one (Figure 1.21).

Fig. 1.20. Architecture of the FaceNet system (from [31]).

Fig. 1.21. 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 [31]).

One thing to mention about the algorithms described above is that they perform the operations on the image of a face. Thus, a face detection is required for previously cropping the face of the person to be identified. Once again, a neural approach can be reduced to an *object detection* problem (detecting the class *face*, in this case). One interesting approach using this technique is *faced* [34]. This is a custom small ensemble of two neural networks, responsible to detect faces and correct the bounding boxes found. The main objective of the system is *speed*, so the main detector architecture is based in YOLO [23], and the second correction stage raises the precision achieved by the detector,

achieving better results than a classical Haar approach, as it can be seen on Figure 1.22. Further comparisons are performed on chapter 3 between these two detection methods.

Fig. 1.22. Classical Haar based face detector [12] (left) vs. *faced* (right). Image from [34].

1.2.3. Embedded deployment

One of the requirements of this work is to be integrated in an autonomous robot. This imposes a power limitation on the algorithms to be deployed. Generally, the robotic systems are deployed using laptops connected to robots, as it was done in [11].

(a) Frontal view.

(b) Side view.

Fig. 1.23. Laptop+robot deployment on [11].

Nowadays, the mentioned increase in the interest into the real-time computer vision applications has fostered the development of specific low-power embedded devices to be integrated in mobile systems. The extending usage of devices such as Arduino or Raspberry Pi has led to embedded robotics systems, such as PiBot [35] (Figure 1.24). These robots are useful in the educational scope, as they are capable of running simple vision and navigation algorithms at a low cost.

Unfortunately, the requirements for running more complex algorithms, such as neural networks, require of the next tier in power terms, keeping the portability nevertheless. The ideal device could be an ASIC, as the custom design would lead to a very tight optimization of the performance. However, the objective is to run the algorithms on

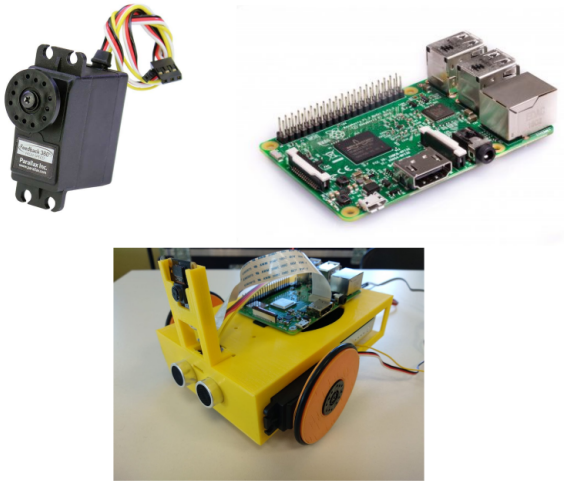


Fig. 1.24. PiBot, an open low-cost robotic platform for education (image from [35]).

existing software frameworks, requiring to use general purpose computers instead. The most remarkable advance in this scope are the Jetson devices manufactured by NVIDIA. These development boards are SoM computers running a tailored version of Linux. The fundamental feature of these systems is that they include a high-performance GPU featuring CUDA, a low-level parallel computation library, as well as several toolkits (such as TensorRT[3]) designed to optimize as much as possible the software implementations for the plethora of possibilities to be designed on this board. As it can be seen in Figure 1.25, its size and power consumption make this system a good choice to be included in an autonomous robot.

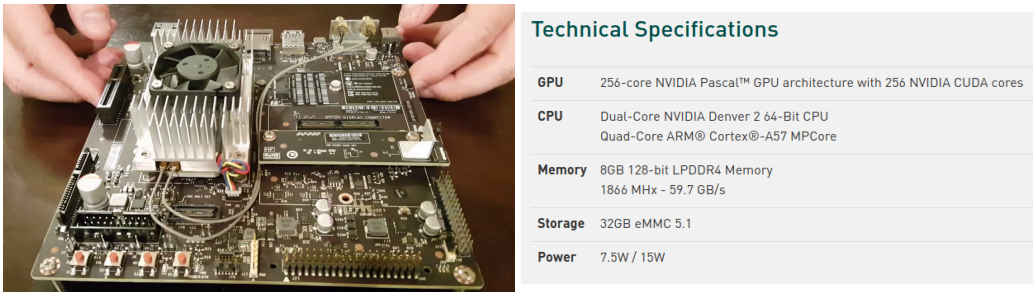


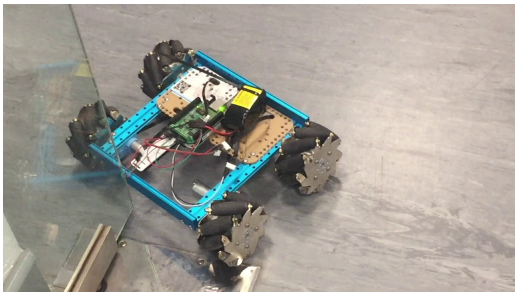
Fig. 1.25. NVIDIA Jetson TX2: an embedded high-performance device including a GPU.

1.2.4. Person following

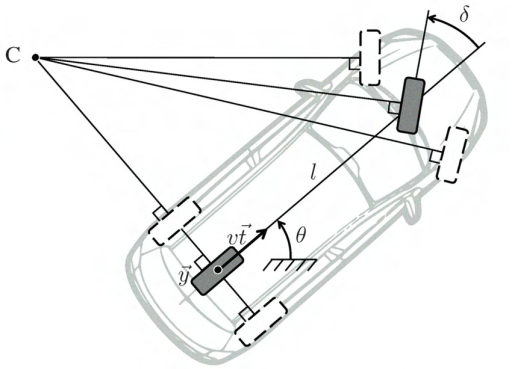
Several approaches have been developed pursuing this challenge of *following a person*. Once the perception algorithms are established, the final output of the pipeline has to be a movement command for the robot to move towards the desired point. Mobile robots can be classified according to their locomotion capabilities. A robot is *holonomic* if the number of its controllable degrees of freedom is equal to its total degrees of freedom.

[3] https://developer.nvidia.com/tensorrt

If the controllable degrees of freedom are lower than the total degrees of freedom, the robot is *non-holonomic*. This difference can be observed on Figure 1.26. In the case of a holonomic robot, the navigation process is simplified, as the robot can instantaneously move to a desired target. However, a non-holonomic robot needs to perform maneuvers in order to move towards a point.



(a) Holonomic robot.



(b) Schematic of the degrees of freedom of a non-holonomic vehicle (a standard car).

Fig. 1.26. Comparison of a holonomic system with a non-holonomic one.

The summary on [36] shows an interesting classification of some existing person following algorithms and their applications (Figure 1.27).

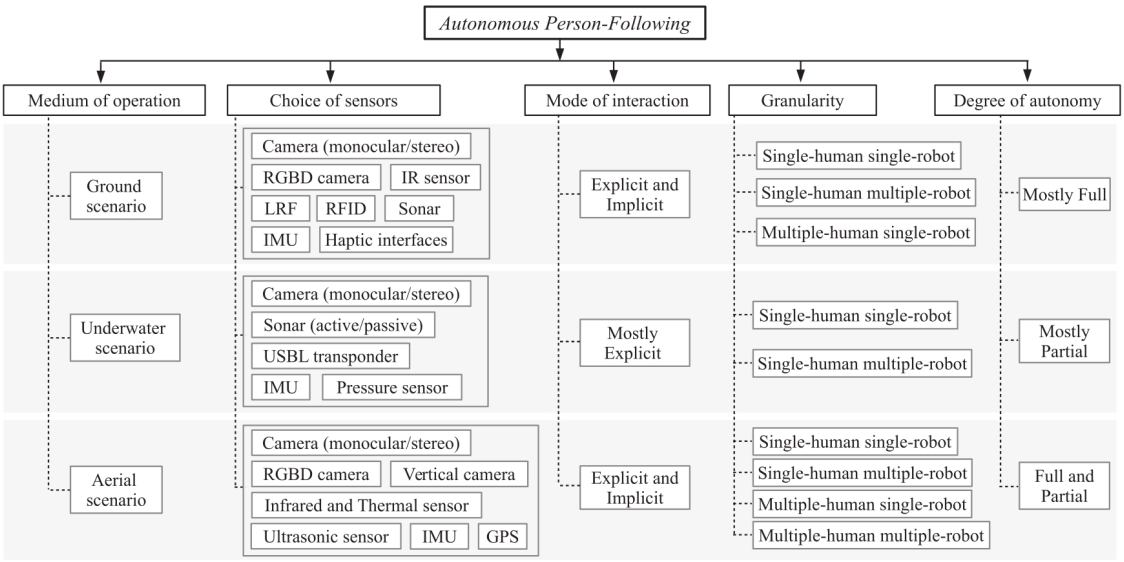
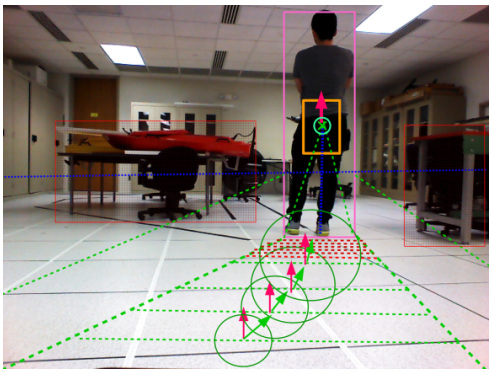


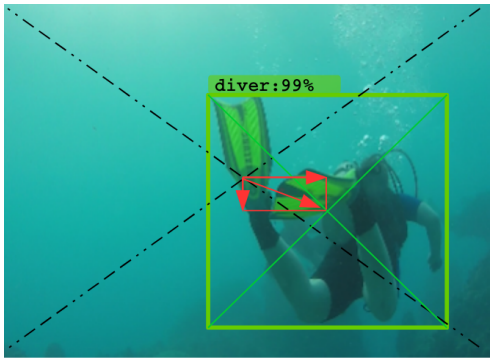
Fig. 1.27. In-depth classification of the existing person following algorithms (image from [36]).

Some approaches leverage the detected objects in order to estimate the relative homography of the orthogonal planes, which allows to partially know the environment of the robot and trace a safe path towards the person, as it can be seen on Figure 1.28a.

Other approaches act without a path planning component, implementing what is called a *reactive* behavior [37], similar to the proposed solution on this work. On these



(a) Following with path computation using homographies (image from [36]).



(b) Example of underwater reactive following (image from [37]).

approaches, the vector between the center of the image and the center of the person is used to command movements on the robot, as it can be seen on Figure 1.28b.

1.3. Objectives

The main objective of this work is to design and develop an embedded system which allows a low-cost robot with a camera to follow a certain person on a robust way. The result will be an autonomous robot which will follow a specific person, whose face has to be known beforehand (using a *reference face* image). This objective, in turn, can be split into specific subgoals:

1. Implement a real-time person following behavior using embedded low-power hardware and a low-complexity educational robot.
2. Build the inference pipeline using exclusively concurrent CNNs (*convolutional neural networks*), as they offer robustness on detection under harsh lighting conditions, such as the ones observed in Figure 1.29.

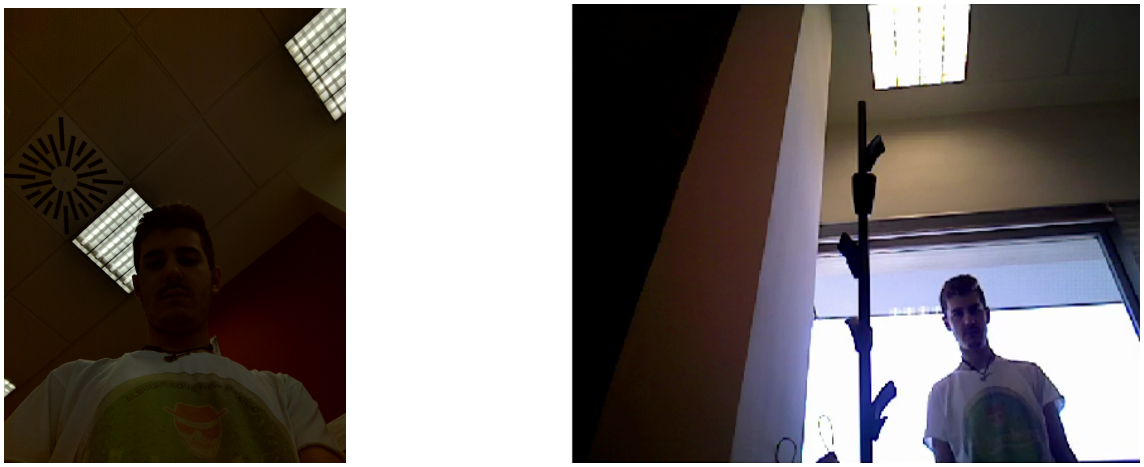


Fig. 1.29. Poor lighting situations for a low-positioned camera.

3. Combine a neural visual perception with optical tracking to carry out a robust following of the persons in front of the robot. This will provide the system with extra reliability and robustness against detection losses/occlusions.

These subgoals allow to summarize the starting point for the development of this project: the available materials are an educational robot equipped with a battery, an embedded *SoM* and a RGB-D sensor.

The structure of this work is organized as follows:

- Chapter 1 presents the motivation of this work, as well as the framework into which it can be placed. Finally, the objectives to be addressed are specified.
- Chapter 2 describes the hardware and software means of the developed work. Later, a full functional description of the implemented system is depicted, describing the

Perception and *Actuation* modules that compose the system. Finally, a description is performed on the software architecture that implements the following behavior and makes the robot to follow the person.

- Chapter 3 describes the experiments conducted on the subsystems and modules of this work. The results of each test are shown as well in order to demonstrate the convenience of the design decisions taken along the development. Finally, a global system experiment is shown, where the robot can be seen on its following behavior.
- Chapter 4 discusses the obtained results on the experiments. Later, conclusions are drawn from the developed work, revisiting the goals and subgoals presented above, and proposing future lines of work that can improve the robot and address its main drawbacks.

2. MATERIALS AND METHODS

This chapter is devoted to describe the process followed to develop the system. The development strategy is based in splitting up the functionality into different modules, which have been tackled sequentially. The next sections will cover each one of the modules, and will describe the solution. Finally, the full ensemble will be described and tested.

2.1. Available materials

2.1.1. Hardware

As it was depicted in section 1.2, typical following approaches work on a personal computer attached to a robot. However, our solution is developed using a devoted SoM: the NVIDIA Jetson TX2, similar to the one described on Figure 1.25. This system features a high-performance GPU, and low-level optimization engines, which greatly reduce the time required to perform the operations required for deep learning applications, such as tensor convolutions. The low power consumption of this board (15W at full power makes it suitable to be embedded in a portable robot equipped with a battery). One drawback of this system is the scarce storage space. However, this can be immediately solved installing an external storage device using the integrated SATA connector it features. In this project, a 120 GB Kingston SSD (*Solid State Drive*) was used for this purpose, leveraging as well on the high transference throughput this device can achieve. It features a 64-bit ARM processor, and it mounts a fully functional Linux system. As it is equipped with two WiFi antennas, a remote control interface can be easily set using SSH connections. Regarding the available RAM in the board, it is limited to 8 GB, to be shared by the GPU and the CPU. This jeopardizes the deployed software and neural networks, which have to be controlled in every moment in order to save the maximum amount of RAM possible.

The vision system used in this work, the ASUS Xtion Pro Live (Figure 2.2), is a USB device composed by a RGB camera and an IR emitter + sensor system, capable of retrieving depth data for each pixel on the image. This is achieved by emitting a known light pattern (Figure 2.3), which reflects in the present surfaces on the scene. These reflections are captured by the IR sensor, inferring the position of the surfaces from the received distribution of the IR pattern.

The last problem to be tackled by this device is the discrepancy caused by the different points of view of the RGB and depth sensor. However, as the distance between these two sensors is fixed and known, a *registration* process can be carried on inside the device,

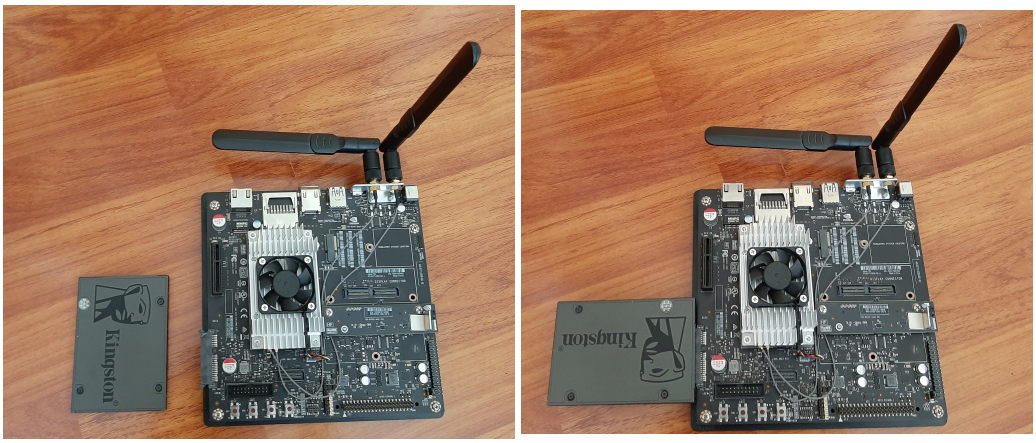


Fig. 2.1. Resulting system: Jetson TX2 board and the installed SSD drive, plugged into the SATA connector.

Fig. 2.2. ASUS Xtion Pro Live

projecting the depth data into the RGB pixels [39].

The systems which implement the described design are called RGB-D sensors, are suitable for robotics, as the yielded result is a point cloud, reflecting the distance from the camera for each pixel in the image. Using this, the device is capable of projecting the 2-dimensional RGB image into the 3D space by means of the depth data (Figure 2.5).

On the other hand, the robot used in this work is the Turtlebot2 educational set. It is based on a Yujinn Robotics Kobuki mobile base (Figure 2.6), which is a non-holonomic robot with 2 degrees of freedom: *linear speed* and *angular speed*.

In the Turtlebot2 version, the mobile base has an attached structure, carrying the RGB-D sensor and a platform where typically a computer can be placed. This platform is useful to mount the NVIDIA Jetson device on. Additionally, as it can be seen in Figure 2.6b, the Kobuki panel is equipped with a 12V output, yielding up to 1.5A, which in power terms can be translated to a maximum power of 18W. As the TX2 board peak consumption is

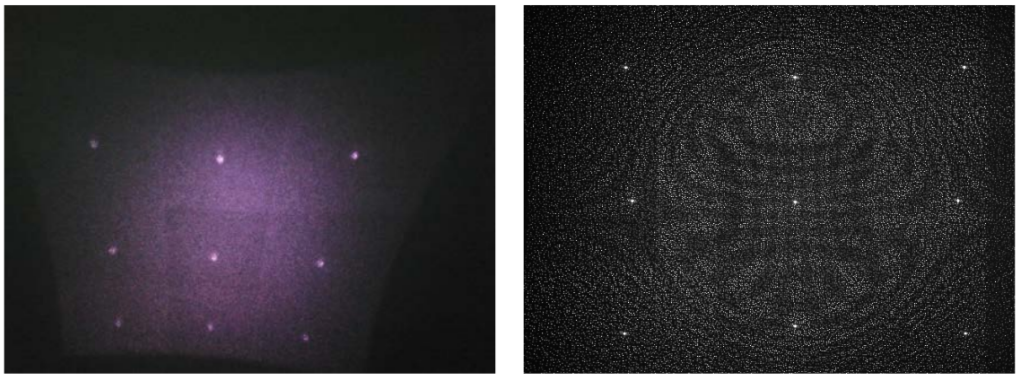


Fig. 2.3. Infrared pattern emitted by the Xtion (images from [38]).

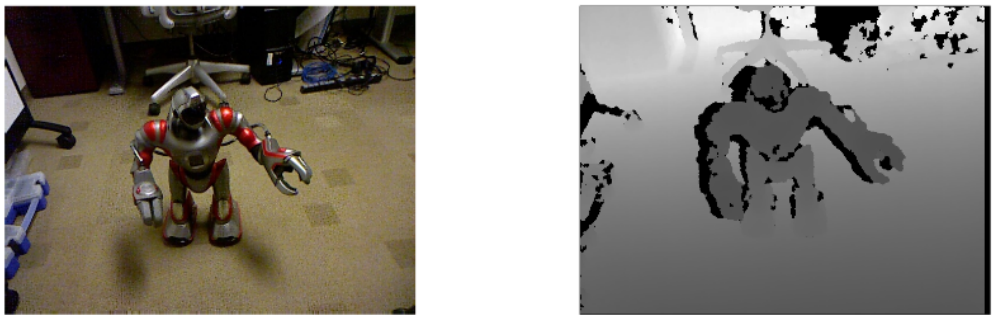


Fig. 2.4. Discrepancy between the RGB and depth images (image from [11]).

15W, this connector is suitable to power the system up, with an additional power margin of 3W. A lookup in the Kobuki user guide [40] allows to find the suitable Molex connector, which can then be attached to two-wire cable and a rounded connector. This provides the NVIDIA Jetson of a 12V DC supply, similar to what it would obtain from a power outlet with a transformer. As the power input is equipped with a DC voltage regulator, it accepts voltages from 5.5V to 19V (table 59 in [41]), this is a successful approach to build an *autonomous* system: powering the computing board from the batteries of the robot, with enough autonomy to be powered on for several hours. The amount of time strongly depends on the usage of the motors of the mobile base, which are the most consuming component of the ensemble.

The final hardware setup is displayed on Figure 2.7, where the described components are combined to build the autonomous setup capable of running high-complexity person following algorithms.

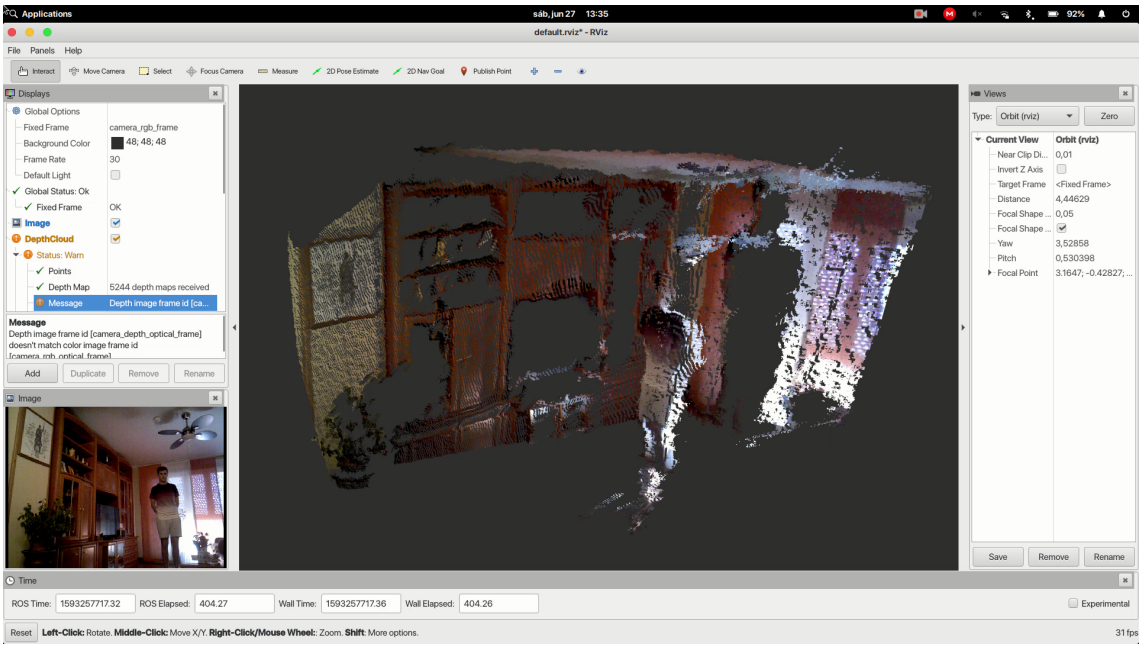


Fig. 2.5. Visualization of the RGB image (bottom left) and the resulting point cloud projected into the 3D space (right).

(a) Appearance of the mobile base robot.

(b) Schematic of the connections panel of the Kobuki.

Fig. 2.6. Kobuki mobile base, which carries the rest of the structure.

2.2. Software

2.2.1. NVIDIA JetPack

The development of this solution has been tackled using exclusively open-source software. The Jetson computing board follows a tightly optimized embedded design guidelines, thus a tailored version of Ubuntu Linux, named NVIDIA JetPack, is developed and maintained by NVIDIA, and it is available for download and install as the board firmware. For the developed system, the version used is JetPack 4.2.2 (R32.2)[4]. This custom implementation includes low-level interfaces for implementing parallel computing operations (CUDA), and several optimizations SDKs (*Software Development Kits*), such as TensorRT. This engine is of special interest for us, as it allows to modify

[4]Details available on: https://docs.nvidia.com/jetson/archives/jetpack-archived/jetpack-422/release-notes/index.html

(a) Front view. (b) Rear view.

Fig. 2.7. Autonomous setup: Turtlebot2 + Jetson TX2 + ASUS Xtion Pro Live.

the underlying implementation of a neural network, swapping certain modules (such as a convolution operation, a ReLU, or an Inception block), for a low-level optimized version of that module, allowing to greatly increase the inference speed without entailing a precision loss. Several details about these optimizations will be depicted later.

2.2.2. Python

This work has been developed using the Python programming language. In the previous work [11], the used version of the language was Python 2.7. However, as of today, that version has reached its EOL (*End of Life*) date, remaining unsupported. For fixing the obsolescence, all the code base was migrated to Python 3.6, a currently supported release, before commiting any change or improvement in the functionality.

2.2.3. ROS

This project requires hardware-software interaction, as the development board needs to read the images received by the Xtion sensor, as well as sending the final velocity commands to the robot. For this purpose, the ROS middleware is used. ROS (Robot Operating System) is *an open-source, meta-operating system for your robot*, maintained by the *OSRF* (*Open Source Robotics Foundation*) [42]. It is a framework that provides a distributed, easily-scalable environment of *nodes*. These nodes are programs which are independently on the computer (or distributed over a 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*. These topics, which rely on a standard TCP/UDP communication between sockets, are intended for an unidirectional, streaming communication, where a node can take roles: *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 is not unrestricted, it must follow a ROS specific syntax, a *Message* type, which is strictly defined for the communication purpose (geometry, sensoring, etc.).

For this project, the packages `cv_bridge` and`openni2_camera` have been used for handling the RGBD data. The robot can be controlled with the package `kobuki_node`. All the software architecture is controlled by `rospy`, the interface for Python to communicate with the described ROS infrastructure.

Another useful feature of ROS middleware is the *ROSBag* storage system. Recording a ROSBag allows to save in a single file the messages read from several topics for the time it is recorded. Later, the ROSBag can be played again to recover the messages from the topics, in the same order they were recorded. This is useful for recording video sequences from the RGBD camera, saving simultaneously the image and depth information, allowing the user to perform testing of different parameters using the exact same image source.

As well as in the Python case, the version of ROS used on [11] reached its EOL date. Thus, the ROS version has been migrated as well to the currently supported release: *Melodic Morenia*, which firstly provided the compatibility with Python 3. As the Jetson TX2 board is based on an ARM architecture, this upgrade has required several tweaks on the software compiling and implementation processes, which have been properly documented in the project repository[5] for the sake of repeatability.

2.2.4. OpenCV

For general image processing, OpenCV (*Open Source Computer Vision*) is a C++/Python/Java open-source library (natively written in C++) for Computer Vision purposes. Among the classic/*state-of-the-art* methods it bundles, several functions can be found, suitable for face recognition, image stitching, eye tracking, computing homographies, establishing markers for augmented reality, etc.

Its general focus centers in *efficiency and real-time functionality*, thank to low-level

[5] https://github.com/RoboticsLabURJC/2017-tfg-nacho_condes

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 even governmental bodies, as their website stands[6]).

This library has been used across the entire project, on its version number 4.2. The It has been useful for diverse tasks, such as image normalization, drawing, computing local features or optical flow approximations.

2.2.5. NumPy

NumPy[7] (*Numeric Python*) is a library for Python (written in C++), born to extend the numerical abilities of this language. It provides a powerful `ndarray` 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 capabilities immediately turn this library into an excellent framework for data processing in a lower level. It allows to store and handle images and tensors on an intuitive way, providing methods to perform typical tasks such as row-wise/column-wise averaging, transposing, type conversing, or matrix slicing. The majority of these structures and methods are implemented using the C++ language, which provides a higher speed than a Python implementation.

2.2.6. TensorFlow

As in the previous work, the deep learning framework used is TensorFlow. This is a high-performance numerical computation library, strongly focused on parallel computing, typically carried on by GPUs or processing clusters. This library is a state-of-the-art means to deploy deep neural networks because of its efficiency. Besides of training/running a neural model, this library allows to load a pretrained model from a storage device, by means of a *frozen graph* file. The information contained in the file are both the network definition and the weights of the nodes.

Additionally, a binding component called `TensorFlowRT/TRT` have been used to implement the low-level optimizations on the TensorFlow neural engines, as it will be depicted later.

[6] https://opencv.org/
[7] http://www.numpy.org/

2.3. Functional architecture

The software implemented in this work has been split into two main components or modules, namely the *perception* module and the *actuation module*, which can be observed in Figure 2.8.

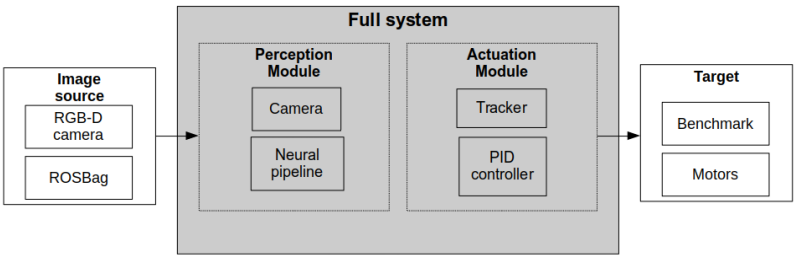


Fig. 2.8. Functional architecture of the developed work, showing the two main blocks.

These two modules cope with specific tasks on an independent manner, as it will be depicted in the following subsections:

2.3.1. Perception Module

This module encompasses what the robot perceives from its sensors (the camera, in this case), and the processing performed to the images in order to determine the location of the person to be followed.

Camera

As it was described before, the Xtion device yields two simultaneous images: an RGB image and a depth image. The ROS controller for the camera, OpenNI2[8], fetches the image and registered depth map from the camera, making this information available through several ROS topics. As ROS follows a *publisher-subscriber* semantics, once the driver is up and running, any application can subscribe to the topics in order to receive all the published messages. In our *Camera* module, two subscribers are deployed to retrieve the latest (RGB, depth) pair on an asynchronous way. These images are then converted into the standard image format in the OpenCV library, and they are ready to be used by other components. Additionally, in order to be able to perform objective testing and benchmarks, the Camera module is able to retrieve the images from a recorded ROSBag instead of the online camera. This is useful to obtain objective metrics of another components of the software on unit testings, as the ROSBag ensures that exactly the same images are used regardless the tested system.

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

The implemented *Camera* module abstracts this condition, allowing to apply the system on an *online* source (camera) or an *offline* source (recorded ROSBag), on a transparent adaptation to the rest of the system. Whenever a new (RGB, depth) pair is required, the *Camera* module will serve the latest available image from the specified source.

Neural pipeline

On the other hand, the received images are passed through an ensemble of neural networks, which provide the capability of detecting the persons in the scene, as well as identifying which one is the one to be followed. As it was studied in section 1.2, the most powerful approaches are achieved nowadays using deep learning. Thus, the complex problem of determining the identity and location of the person of interest has been decomposed into three tasks, which are addressed using deep learning techniques:

1. **Person detection:** the generalized *object detection* task (Figure 2.9) is a commonplace challenge in computer vision. The existing solutions use object detectors similar to those explained in section 1.2, which are typically trained in large image datasets. The classes these models are capable to detect contain the *person* class. Thus, as it was demonstrated in [11], a deep object detector can be utilized for detecting persons inside the image. On this work, several models have been tested, varying the base network architecture and depth. As one of the objectives of the system is to work on a portable (low-power) system, only the architectures which yield a good performance with a sufficiently low inference time are considered. The two most suitable models for this purpose are SSD [19] using a MobileNet [21] for feature extraction, and YOLOv3 [25] on its tiny version[9]. These models are previously trained and publicly available on the TensorFlow Model Zoo [43] and on repositories hosted on GitHub[10]. In-depth testings have been developed to compare the performance of these two models, which can be found in chapter 3. The previously developed work [11] only supported SSD-based detectors, however, the object detection component of the program has been upgraded and it features YOLOv3 support as well, making it available through the configuration file specified on launch.

2. **Face detection:** as in the previous point, the problem can be addressed using an object detection neural network. However, the previously deployed models are not capable of detecting faces. As the system is power-limited, looking for another multi-class model could be considered as an overshooting solution that the system

[9]The usage of the tiny version of YOLOv3 is due to issues with the limited memory on the Jetson TX2 board. The full model was tried unsuccessfully, as it requires more memory than the available one on a typical execution.

[10]https://github.com/mystic123/tensorflow-yolo-v3

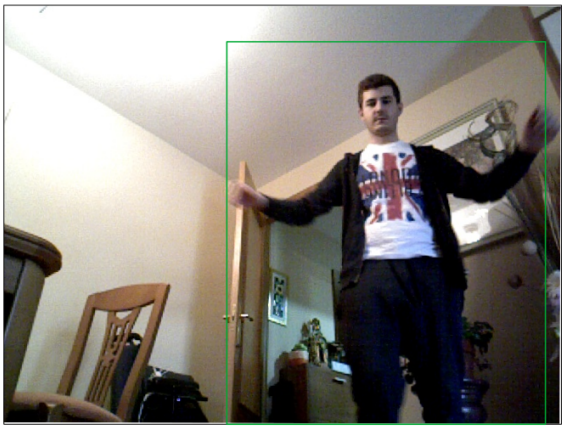


Fig. 2.9. Example of a person detection task.

is not able to handle at the same time the other model is deployed. Thus, one feasible solution is to use a single-class detection system. The network trained in [34] implements a two-stage neural network capable of detecting faces. As it was depicted in section 1.2, this detector is based on YOLO, which ensures a high-speed and efficient detection deployed using a class-specific neural network, which is way lighter than a multi-class detection system. The repository where the project is hosted contains a video sequence comparison comparing the precision of this system against a classical Haar cascade approach [12]. Chapter 3 contains captions of this sequence to show the superior performance for the face detection issue.

3. **Face identification:** Once the face of a person has been retrieved, it can be used as a distinguishing feature for determining their identity. As the basis of this work is to take advantage of deep learning power, a neural system has been selected to perform this task. For this purpose, *FaceNet* (described on Figure 1.2.2) has been deployed in order to perform identification, using a publicly available implementation in TensorFlow[11]. As a result, the image of a face is transformed into a 128-dimensional vector: its projection or *embedding*. This transformation is learned after a triplet-loss training process, which separates different faces as much as possible, while projecting similar faces as close as possible. As it can be seen on Figure 1.19, this can be use to obtain similar projections when two images of the same face are evaluated, despite different lighting conditions (as a channel-wise normalization step is performed before passing the image through the network).

To sum up, this ensemble of 3 neural networks provides a sequential pipeline to obtain *person locations, face detections and face projections* from a single image, taking advantage of the flexibility that deep learning methods offer, in order to address three different problems on an efficient way. Its functionality has been depicted in Figure 2.10.

Once the inference pipeline has been designed and implemented, it can take advantage

[11] https://github.com/davidsandberg/facenet

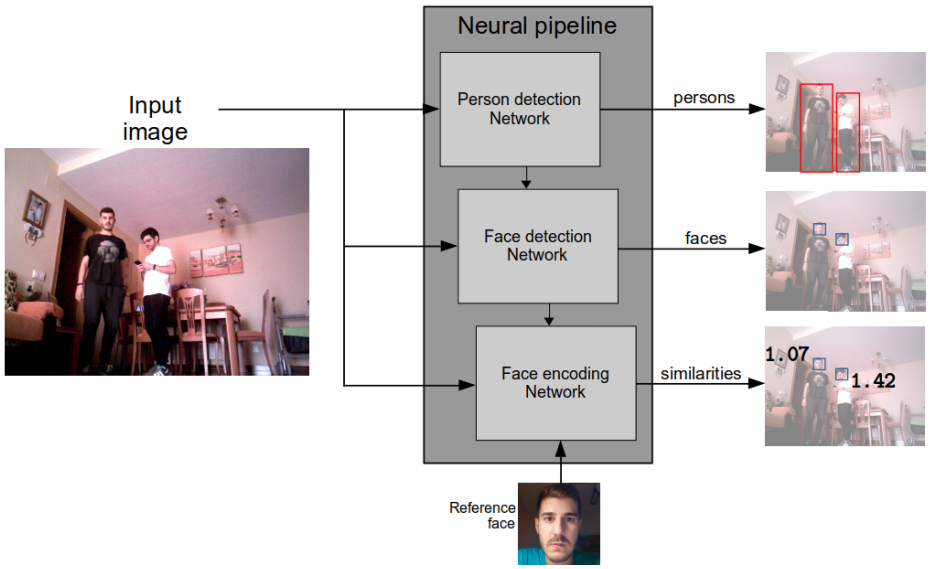


Fig. 2.10. Neural pipeline, showing the cascade of the three neural networks used to output persons, faces and similarities with the reference face.

of the optimization libraries present in the Jetson TX2 board, using TensorRT for this purpose. Using this library, several segments from the architecture of a given network can be modified according to certain parameters:

MSS (*Minimum Segment Size*) the threshold above which a segment is selected to be replaced by the TensorRT optimization. Increasing this value makes the optimizer more selective, in order to optimize only the heaviest segments of the network. A low value aims to optimize smaller segments, although this can cause an excessively high overhead, causing the resulting graph to run slower than the original one.

MCE (*Maximum Cached Engines*) TensorRT keeps a cache of engines on runtime, with the purpose of reducing the time spent for loading them into the GPU. This parameter modulates the amount of engines kept in that cache, in case the available memory to establish the cache is limited.

Precision mode typically, the weights and parameters of the trained neural networks are handled as 64-bit floating point numbers. A reduction in the precision to 32-bit or 16-bit achieves very similar results, making the operations much lighter as the precision mode is reduced by $\frac{1}{2}$ or $\frac{1}{4}$. A more daring approach reduces the precision up to 8-bit integers, performing an additional *quantization* step since the range will be limited to 256 values. The quantization step analyzes the segment, computing the numeric range of its weights. This range is typically narrow enough to perform a 8-bit quantization, mapping the high-precision weights into a range composed of 256 values between the minimum and maximum values of the weights.

An experimental tuning of these parameters has been performed in chapter 3, looking

for an optimization of the inference time and taking into account that the enhanced models of the three neural networks have to share the limited memory on board. Thus, care has been put into the memory footprint that an excessive runtime optimization might cause, as this leads to a high time penalization if the system cache is utilized to store the models.

The *Camera* and *Neural* components form the *Perception* module, responsible of retrieving the external image and apprehend the underlying information from the image: position and identity of the person to be followed. This information serves as input to the actuation module, explained below.

2.3.2. Actuation Module

The second module of the system addresses the actuation task: once the external stimuli have been acquired and processed, an action has to be performed in order to move the robot towards its goal. As the final objective of the system is to follow a person, these movements have to be reactive, happening as soon as possible whenever the person changes their position.

Motion Tracker

The previously depicted *Neural* component outputs trustworthy inferences with a certain refresh rate, namely k, which can reach a relatively high value depending on the current load and power profile in the development board. If k is too high, the system can run the risk of suffering an important delay when the movement is performed. This can lead in unsteady movements, with a higher probability of losing the reference person. To avoid this, a *Tracker* component is added to the system. Its functionality is to be able to *estimate* the person movement during k frames, while the neural pipeline is performing detections. This way, currently detected persons can be tracked along the image while they wander, until the neural ensemble outputs the latest predictions, which determine the true new position of the persons. To fulfill this requirement, the tracking method has to be able to run at a higher rate than k, preferably with a considerably lower inference time. This way, the system counts on a slow, reliable detection system supported by a fast estimation system, devoted to guess the movements between detections.

The method chosen for this purpose is a *Lucas-Kanade* visual tracker [44]. This technique estimates the *motion field* between the images taken in two time instants, addressing the problem using a differential approach [45]:

The basis relies on the fact that in a video sequence, for small changes in space and time, the intensity remains constant inside a certain pixel neighborhood:

$$\mathbb{I}(\mathbf{x}, t) = \mathbb{I}(\mathbf{x} + \Delta\mathbf{x}, t + \Delta t)$$

Using a 1st order Taylor series approximation and algebra, the *optical flow equation*, f, can be found[46]:

$$f_x u + f_y v + f_t = 0$$

where

$$f_x = \frac{\partial f}{\partial x}; f_y = \frac{\partial f}{\partial y}$$
$$u = \frac{dx}{dt}; v = \frac{dy}{dt}$$

f_x, f_y and f_t represent the image gradients with respect to the space and time, respectively, and (u, v) represents the movement vector of the scene.

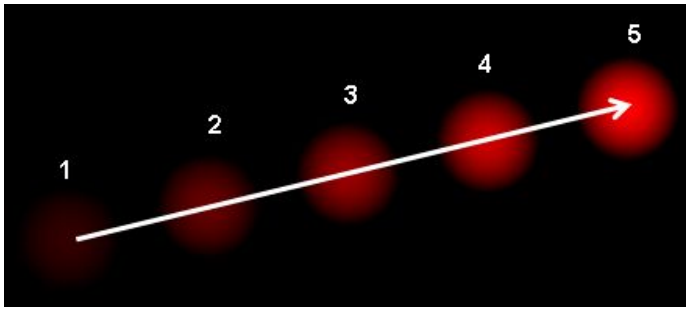


Fig. 2.11. Optical flow for different time instants. Image from [46].

At this point, the resulting system is under-determined as the problem presents 1 equation with 2 unknown variables. Lucas-Kanade addresses this problem taking advantage of one of the previous assumptions: as the pixel intensity remains constant for a pixel neighborhood, one can expect the same movement on neighboring pixels: these will share a common (u, v) movement vector (typically, a small square or circular neighborhood is assumed). Assembling together those equations results in an over-determined system, where a *Least-Squares* solution yields the best-fitting movement vector (u, v) for that neighborhood, allowing to have a local estimation for the movement in that area:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i {f_{x_i}}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i {f_{y_i}}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix} \tag{2.1}$$

The computation of Equation 2.1 can be efficiently performed with high-performance libraries, such as *NumPy* or *TBB*, which ensure a fast execution of the estimation. This makes Lucas-Kanade estimation an efficient approach to compute the optical flow in tasks such as image registration, video stabilization or depth computation in stereo vision systems. This technique is implemented in the *OpenCV* library through the method

`cv2.calcOpticalFlowPyrLK`, which evaluates the image on a pyramid of scales to improve the robustness. This method offers a set of tunable parameters to detect the new position of the corners:

`winSize` size of the window to perform the LS solution.

`maxLevel` number of additional scales to evaluate the image on a pyramidal scale sequence.

`criteria` flags to determine the stop condition on the iterations of the algorithm.

However, in the case of study of this work, a calculation of the motion field on the entire image would be considered overshooting (besides of a dangerously slow task for a real-time system), as the objective is not to compute the entire optical flow. The estimation can be limited to the pixels inside and surrounding the persons in the scene. Furthermore, one can notice the existence of more informative regions inside the person than others, given its texture: typically object *corners* will be the best choice to be tracked [15], given their easiness to be identified and the fact that they provide more motion information than another areas (aperture problem). In order to detect these corners, a Harris corner detector can be used, analyzing the eigenvalues inside different 2-D windows inside the image. A *corner response* can be computed, yielding a score depending on the eigenvalues and their ratio:

$$R = \det M - k(\text{trace}(M))^2$$

with k being an empirical constant $k = 0.04 - 0.06$, and M being the diagonal matrix resulting of the singular value decomposition of the current window.

The value of R determines the decision taken of the window containing a corner.

A modification of this algorithm, the *Shi-Tomasi* corner detector was published on [47], improving the performance of the corner detector by changing the corner response computation to:

$$R = \min(\lambda_1, \lambda_2)$$

taking the window as a corner if R is greater than a given threshold. The scoring diagrams for determining the corner response on the two depicted methods can be observed in Figure 2.12. One advantage of this methodology is its invariance to rotation, as it works using the eigenvalues, that automatically align to the most variant directions. However, one important thing to mention as a flaw is the variance to scale: the relative size of the corner with respect to the window size has influence on the eigenvalues.

Other methods for corner detection are widely used in state-of-the-art developments, such as SIFT [48] or FAST [49]. However, according to the evaluation among several corner detectors on [50] reveals that the Harris/Shi-Tomasi approach yields a more reliable result for this purpose, while taking a low time to execute: it takes around 25 ms to

evaluate the 640×480 image from the Asus Xtion, which makes the tracking module to run 5x faster than the neural pipeline.

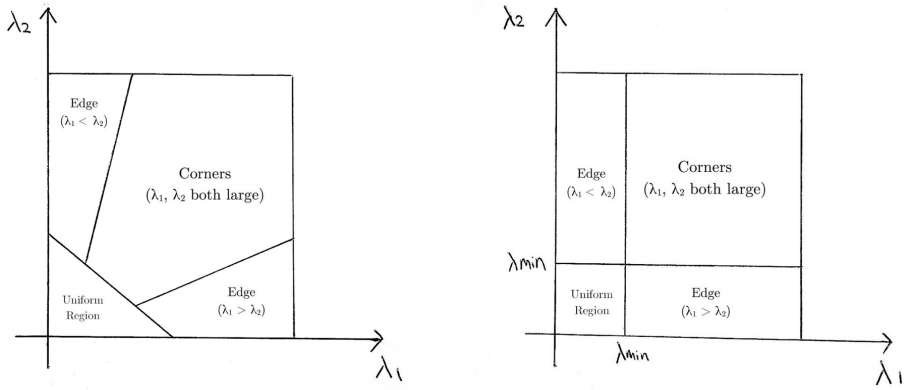


Fig. 2.12. Corner response R scoring functions on $\lambda_1 - \lambda_2$ on the Harris (left) and Shi-Tomasi (right) detectors (source:[51]).

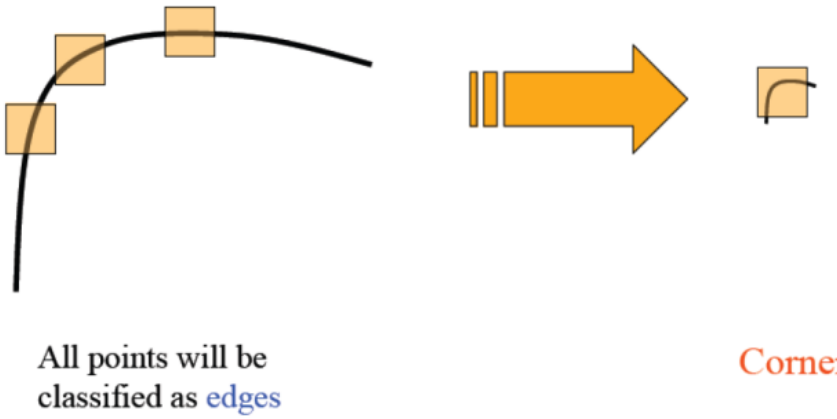


Fig. 2.13. Scale variance of this method, where the size of the corner with respect to the `winSize` jeopardizes the eigenvalues.

Using this method returns what the authors call the *good features to track*, namely, the best N corners of the image or region provided.

This method is implemented in the *OpenCV* library through the method `cv2.goodFeaturesToTrack`, which offers a set of tunable parameters to extract corners from a given image:

`maxCorners` maximum number of corners to be found.

`qualityLevel` multiplicative factor for the R of the best corner. A corner response below qualityLevel $\cdot R_{max}$ will be discarded.

`minDistance` minimum euclidean distance between the selected corners.

`blockSize` size of the pixel block to compute the eigenvalues.

The combination of these two methods provides a fast methodology to estimate the movement of a region using exclusively algebraic calculations on the pixel intensities. As these computations are bounded in complexity, the iteration time is around 5x faster than the neural pipeline. Thus, the simultaneous combination of both algorithms allows to track the movements of the persons during K frames, until the next neural update arrives. This is shown in Figure 2.14.

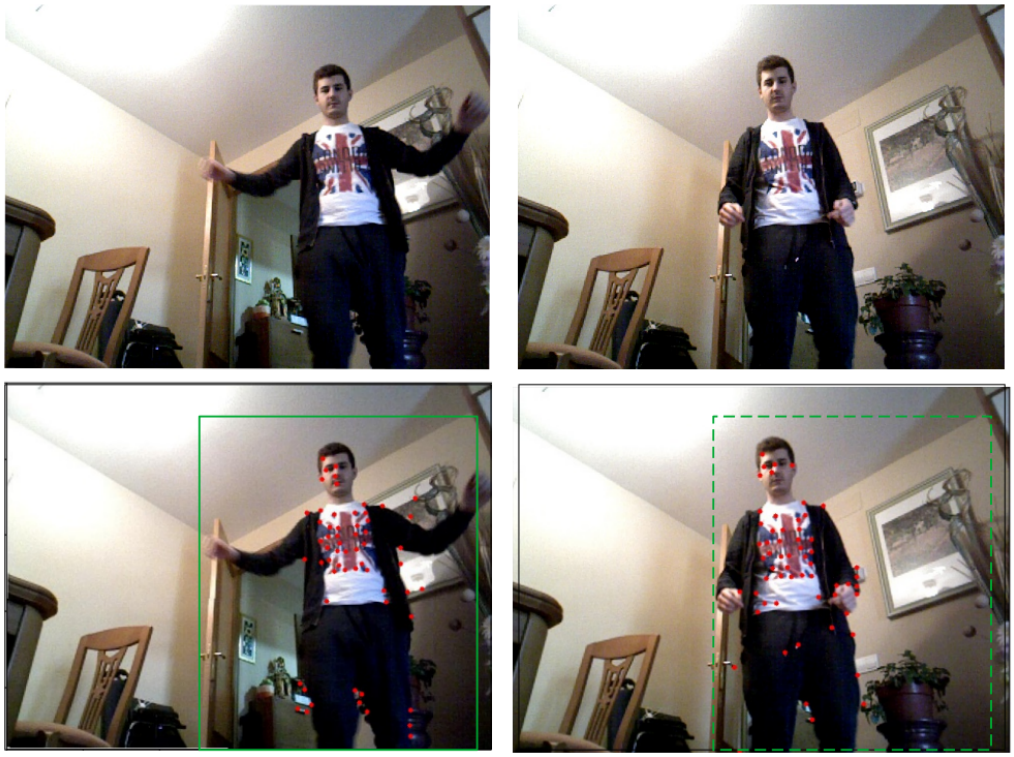


Fig. 2.14. Operation of the tracking module: the last detection (green) determines the person position. The keypoints (red) are tracked during K frames until the next neural update.

As the *OpenCV* implementation of Lucas-Kanade identifies the points that have been found in both frames, the average displacement of all the points can be computed. This allows to displace the bounding box of that person using the computed displacement vector. This is required, as the bounding box changes its position and size when the person moves, as Figure 2.14 shows. Additionally, it can be rescaled in case the person moves closer or further from the camera, using the distribution of the points in the previous and current frame. As it can be seen on Figure 2.15, the Shi-Tomasi corner detector finds a set of corners (keypoints) in the frame t, drawn with red points. These points are distributed with a given mean: , the centroid of the cloud, represented with an "x", beside of a standard deviation pair (σ_x^t, σ_y^t). On the next frame, some new keypoints are found (green), whereas keypoints from the previous frame are identified. These points are useful for computing the new centroid $(\mu_x^{t+1}, \mu_y^{t+1})$ and deviations pair $(\sigma_x^{t+1}, \sigma_y^{t+1})$. With this information, the person box can be updated accordingly:

$$\text{person\_coordinates}(t) = \left[\mu_x^t, \mu_y^t, w, h\right]$$

$$\text{person\_coordinates}(t) = \left[\mu_x^{t+1}, \mu_y^{t+1}, w \cdot \frac{\sigma_x^{t+1}}{\sigma_x^t}, h \cdot \frac{\sigma_y^{t+1}}{\sigma_y^t}\right]$$

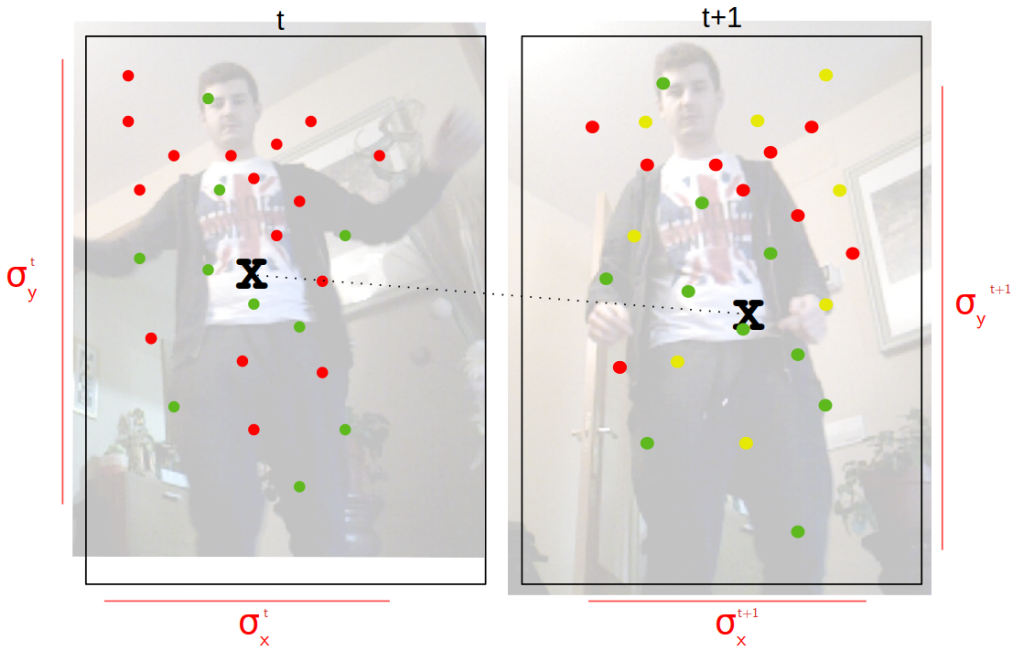


Fig. 2.15. Update of the Lucas-Kanade tracker from frame t to frame $t+1$. The green points are the correctly detected in both frames, while red and yellow points are only detected in t and $t+1$, respectively. The green points determine the new centroid and the shape deformation of the box.

This way, the update is sensitive to displacements and scale changes in both directions, in case the person changes their linear distance to the camera.

Its introduction enhances the robustness as the output of the system will not depend just on the neural detections. This improves the performance as partial occlusions might cause some detections to be discarded momentarily. The introduction of the tracker can palliate this effect, as the person will be kept as *detected* for a number of frames even if it is not detected by the neural pipeline, and its position will be tracked using Lucas-Kanade. This number of frames is called *patience*, P, and introduces an hysteresis in the tracker, as a person has to be lost for P frames in a row to be discarded.

On the same way, a detection has to be maintained during P frames to be joined to the tracked persons. The patience component is introduced in pursuit of stability in case the scenario complicates a stable detection. In such cases a detection flickering is observable,

and this could lead in an erratic movement on the robot. The introduction of the patience solves this problem successfully.

PID Controllers

The combination of the described systems results in a efficient way to detect and identify the person to be followed, and additionally, track their movements on a fast way between slower detections.

The last block of the system is responsible of translating this location information of the reference person into velocity commands that move the robot towards an *acceptable position* with respect to the person, where certain conditions are fulfilled.

As it was described on section 2.1, the robot offers 2 degrees of freedom: rotation speed and linear speed. Thus, this *acceptable position* can be described in those 2 dimensions:

Angular position: the reference person has to be placed at a side angle of 0º with respect to the robot front.

Linear position: the reference person has to be placed at a distance of 1 m with respect to the robot front.

Due to the sensors uncertainty, the prediction and tracking estimation, and the movements of the person, these positions have to be extended to *safe areas*, inside of which the robot will not trigger a velocity command for that dimension. This is achieved introducing a *margin/tolerance* on each dimension. Additionally, these geometric criterion have to be translated to measurable discrepancies. This way, the safe zones can be defined as:

Angular zone: the reference person has to be placed at the horizontal center of the image, with a margin of ±50 pixels on the sides.

Linear zone: the reference person has to be placed at a distance of 1 m with respect to the robot front, with a distance margin of ±30 cm[12].

These regions, which are completely tunable using the configuration file, can be visualized on Figure 2.16.

[12]This criteria can be maintained in metric distance, as the depth sensor specifically yields that information. In the angular case, the image is a 2D projection on the camera plane, which does not allow to infer the relative angle with the person without extra computations using the relative distance.

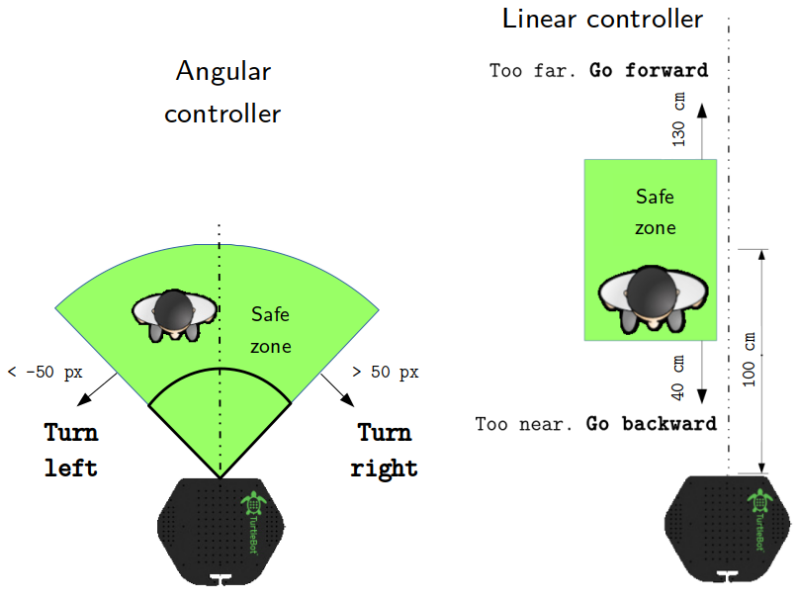


Fig. 2.16. Safe zones for each controller. Image from [11].

To place the person inside these safe zones, the robot has to move on certain directions. For determining a movement, an *error* vector (e_x, e_w) is computed, using the tracked person coordinates:

e_x**:** the linear error or *range* is computed using the depth image, estimating the distance from the robot to the person. As the Xtion sensor registers the depth image into the RGB one, the person coordinates can be used in the depth image in order to find the distance of each pixel inside the bounding box of the reference person: the *person depth map*. As it is feasible that the box contains an important region of the background (specially if the person opens their arms, as the neural detection will encompass the entire body), the edges of the depth map are trimmed. Later, a 10x10 grid is computed to have 100 uniformly distributed samples of the depth of the person. In order to ensure that the background does not affect the range measurement, the median value is computed, as even if some outlier points belong to the background, they would have to make up the 50% of the sampled set to deviate the measurement from the true range.

e_w**:** the angular error can be computed taking into account that if the robot and the person are aligned, its bounding box will be horizontally placed near the center of the image. Therefore, an error metric can be extracted computing the difference on the horizontal coordinate between the image center and the center of the bounding box of the reference person.

These computations can be visualized on Figure 2.17.

The last step of the controller takes care of computing two proper response (linear and angular) for the robot. If these responses depend only on the error readouts, the robot

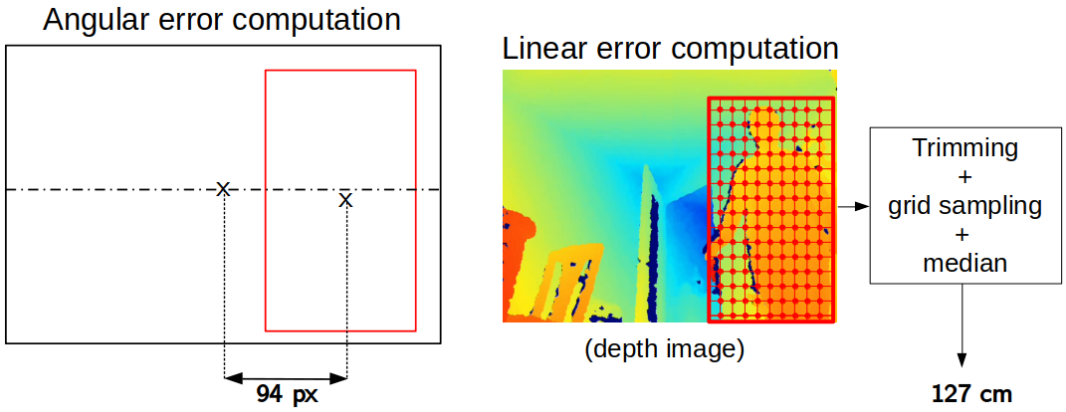


Fig. 2.17. Error computation on each controller.

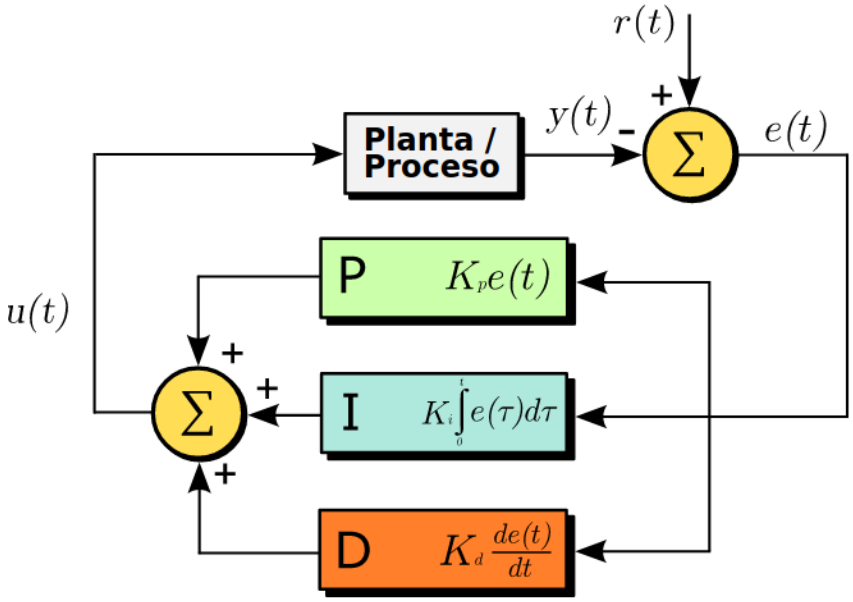


Fig. 2.18. Schematic of a generic PID controller.

might receive unsteady commands, that might cause a total loss of the person from the field of view. This can be solved introducing a slightly more complex system: a PID controller [52], which is a closed-loop control system that outputs a response taking into account the previously sent responses.

The *PID* acronym stands for *Proportional, Integral and Derivative*, as that is the methodology followed to output a response. The output in the time instant t, $u[t]$ depends on the currently measured error, $e[n]$, and it is computed as it can be seen on Figure 2.18:

This can be expressed by means of the following equation:

$$u[n] = k_p e[n] \quad + \quad k_i \sum_{i=0}^{n} e[i] \quad + \quad k_d(e[n] - e[n-1]) \tag{2.2}$$

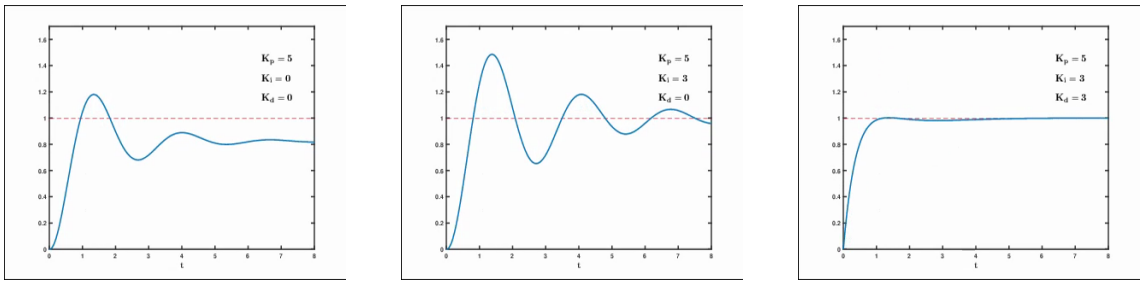
This equation can be split into the three components:

***Proportional*:** $k_p e[n]$. This is the basic component, that computes a response directly proportional to the measured error.

***Integral*:** $k_i \sum_{i=0}^{n} e[i]$. An additional response, equivalent to the sum of the total error until the current instant. 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, increasing the response magnitude in order to close the existing gap between the error and the desired readout[13].

***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[14]. 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.

Figure 2.19 shows that the combination of the three sub-responses can achieve a fast and steady response (Figure 2.19), bringing back the system under control on an efficient way.



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

Fig. 2.19. Different controllers response along time.

Each contribution is parameterized by its corresponding constant (k_p, k_i, k_d), so a task to perform is to find the optimum value for each one of them. Visual assessments of the robot stability under different combinations lead to the values present in Table 2.1, which yielded a steady behavior of the robot when it is subject to typical indoor conditions of following a person wandering. As previous parameters, these values can be changed using the configuration file.

Finally, when the speed is computed, it is adapted to a ROS `Twist` message, and it is published to the topic devoted to velocity commands to the robot. On the other side

[13] When the monitored variable goes into the tolerated zone again, the total error has to be reset, as it is not required from now on.

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

	Linear	Angular
k_p	0.4	0.005
k_d	0.04	0.0003
k_i	0.05	0.006

Table 2.1. Optimal found values for the parameters in each PID controller.

of the topic, the driver reads these messages and moves the robot accordingly with the commands received.

This last block completes the functional architecture of the full following system.

2.4. Software architecture

The developed software puts all the previous components together, offering two application modes:

`followperson` mode: this is the default functionality of the system. When running on this mode, the program feeds the tracker and the neural pipeline with images from the ASUS Xtion, and sends the velocity commands to the robot, writing them into the specified ROS topic.

`benchmark` mode: this mode is designed to test the entire infrastructure, with the purpose of tuning parameters or extracting objective metrics for comparisons, such as precision, or inference time. The images are read from a previously recorded ROSBag, emulating the Xtion sensor and providing always the same RGBD sequence to be fed in different implementations, allowing to compare the performance of different configurations under identical conditions. On this mode, the velocity commands are not sent to the robot, just drawn in the output image (Figure 2.21), which is also saved into an output video for later visualization. Aside of the video, execution graphs and YAML[15] files are stored containing information about the tracked persons and times for each frame processed by the `Main` thread.

This behavior, and other parameters, can be configured on the program execution without modifying the source code. The program receives a YAML configuration file specifying all the required parameters in order to run the system:

```
1 NodeName: "followperson"
```

[15] YAML is a plain-text data serialization format. It has been chosen as a standard format on this project as it offers a good tradeoff between serialization (allowing the data to be converted back into data structures in Python) and readability of the file without processing it.

```
Benchmark: true # true for benchmark, false for followperson
RosbagFile: "resources/bag1.bag"  # path to the ROSBag if benchmark
LogDir: "resources/benchmarks" # where to write the results

Networks:
  # Parameters for the neural pipeline
  Arch: ssd # detection architecture [ssd, yolov3, yolov3tiny]
  DetectionModel: "models/ssd_mobilenet_v1_0.75_depth_coco.pb"
  DetectionWidth: 416 # usually 300 for SSD, 416 for YOLOv3tiny
  DetectionHeight: 416 # usually 300 for SSD, 416 for YOLOv3tiny
  FaceEncoderModel: "models/facenet_inception_resnet_vggface2.pb"

RefFace: "resources/ref_face.jpg" # Image of the reference face

Topics:
  RGB: "/camera/rgb/image_raw" # topic publishing the RGB images
  Depth: "/camera/depth_registered/image_raw" # topic publishing the
      depth images

# Parameters for the speed controllers
XController:
  Kp: 0.4
  Ki: 0.04
  Kd: 0.05
  Min: 0.7
  Max: 1

WController:
  Kp: 0.005
  Ki: 0.0003
  Kd: 0.006
  Min: -50
  Max: 50
# Parameters for the people tracker
PeopleTracker:
  Patience: 5
  RefSimThr: 1.0
  SamePersonThr: 60
```

The previously depicted structure can be implemented on the Jetson board using the programming language Python. As the tracking module has to run asynchronously, the `threading` library is used, deploying the following threads:

Main thread: the purpose of this thread is to continuously draw the output image (shown in Figure 2.21 and explained below), and compute the errors and suitable responses, as well as sending them to the robot. One thing to notice about this thread is that it does not process all the frames in the sequence, as its rate depends on the drawing time and the computation of the response. It works asynchronously, fetching the latest frame from the `tracker` thread.

`networks_controller` thread: this controller handles the 3 described neural networks, running sequential inferences on them. In the Jetson platform, these neural networks are deployed in the GPU of the board. Therefore, this thread can be seen as the one which interacts with the GPU in order to pass, retrieve and transform tensors from the networks.

`tracker` thread: as it was shown before, the tracker must inherently iterate at a higher rate than the neural infrastructure. However, including it in the main thread would be bad for its performance, as the speed would be limited by the image drawing and responses publication in the speed topics. Therefore, it is extracted to a devoted thread. The simplicity of the Lucas-Kanade tracker makes it fast to execute, however it would be pointless to track a person several times before a new image arrives from the camera. To avoid this, the thread has a rate limitation of 30 Hz, equal to the framerate of the Xtion sensor.

As this is the fastest thread to execute, and it is crucial that the tracker has available every image from the camera, this is the first component to receive the images from the source, on a 30 Hz synchronous manner. The rest of components can fetch the images asynchronously from the tracker whenever they need them.

ROSCam: this component, responsible of fetching the images from the source (a ROSBag or the Xtion camera, as explained before) is not deployed as a thread. However, as it works by means of subscribers when a synchronous mode is required (thus, when the source is the Xtion camera), the ROS API for Python, `rospy` automatically deploys these subscribers on independent threads.

This software architecture can be seen in Figure 2.20, where the interaction behavioral between the threads can be visualized. The `Main` thread varies its behavior depending on the configured mode (`followperson`/`benchmark`), whereas the rest of threads behave similarly in both configurations.

The visible output of the system is the image shown in Figure 2.21. This image is drawn by the main thread, when the position errors are computed and the responses have been sent to the robot, and it serves as monitoring means for the execution, showing the images, the tracked persons and the sent commands. If the benchmark mode is enabled, these image are appended to a output video, which serves for posterior visualization or assessments of the performance.

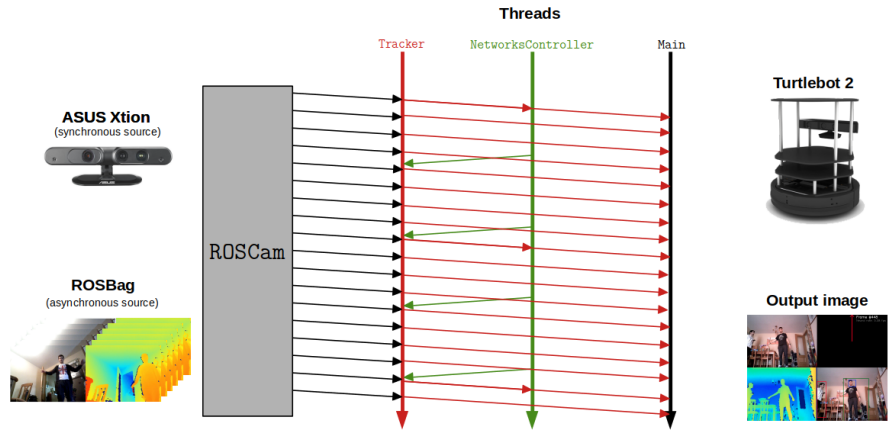


Fig. 2.20. Software architecture for the system.

Fig. 2.21. Output image drawn by the program. Upper left: input RGB image. Bottom left: input depth image. Upper right: velocity commands sent to the robot, and information about the neural rate and number of current frame. Bottom right: tracked persons (green if it is reference, red otherwise) and their faces

3. RESULTS

This chapter describes the different experiments and benchmarks applied to the proposed system and its subsystems. These test have the purpose of taking design or implementation decisions, selecting the best choices to measure the achieved performance, accuracy and robustness of the final system and subsystems. For this purpose, several video sequences were recorded with the ASUS Xtion inside ROSBag files. This way, the same video can be used to assess the performance of different configurations, ensuring that the results will not be affected by external variability due to different environmental conditions on the test data.

The majority of the tests described below for the neural pipeline measure the IoU score, which determines the overlapping quality between two bounding boxes. Thus, it is required to label the video sequences, specifying on each frame the location of the ground truth labels for every video. For this purpose, the tool LabelMe [53] was used to provide the labels to the video, creating a JSON file for each frame of the video sequence. A screenshot of this tool is shown on Figure 3.1.

The source code of the experiments conducted below can be found in a separate `experiments` branch of the source repository on GitHub[16], hosting both the testing and plotting source files, as well as the CSV files containing the data plotted in the figures below.

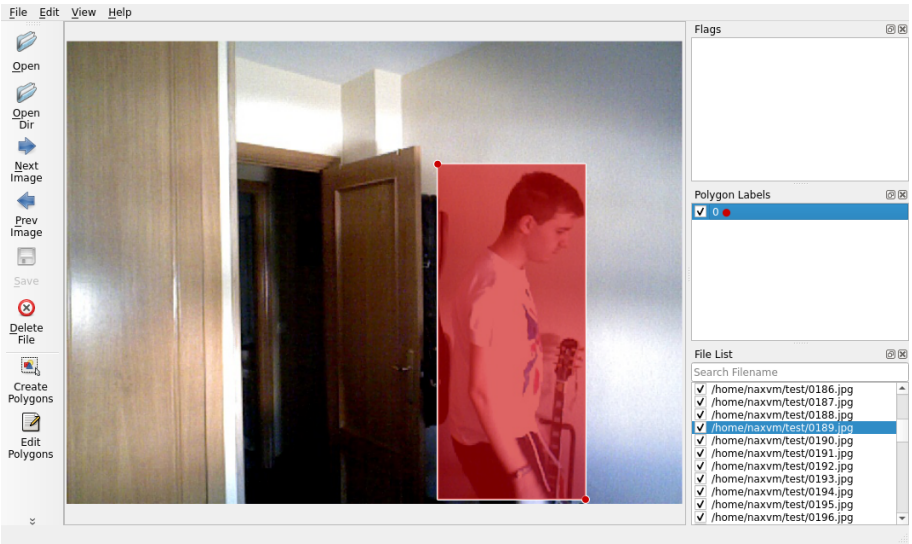


Fig. 3.1. Interface of the LabelMe annotation tool [53].

[16] https://github.com/RoboticsLabURJC/2017-tfg-nacho_condes/tree/experiments

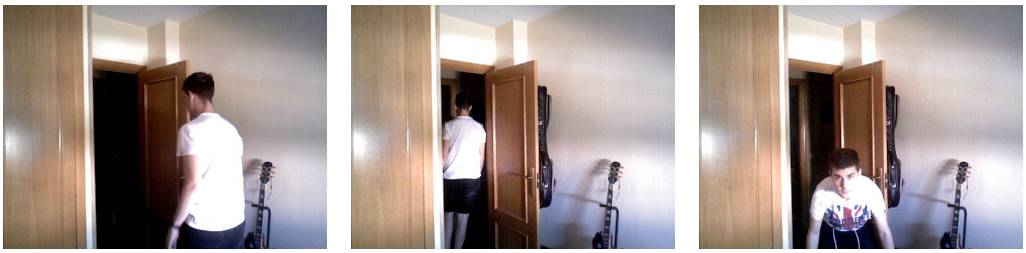
3.1. Person detection experiments

This experiment compares the two detection architectures implemented on this system: YOLO [25] and SSD[19].

In the case of YOLO, the implemented architecture is YOLOv3, in its *tiny* version. This is due to the memory constraints of the Jetson board where the models are loaded. The available memory (8 GB) has to be shared among TensorFlow and the rest of processes, causing the more memory-intensive models to fail on loading. The YOLOv3 full model demands too much memory, making impossible to use it properly on the Jetson TX2 board. Thus, the chosen architecture is a lighter one, publicly available on the YOLO website[17]: the Tiny YOLOv3 model.

On the other hand, as it was explained in section 1.2, on a real-time application the most convenient variant of the SSD-based detectors is the one that uses a MobileNet [21] as a feature extraction network. The TensorFlow Model Zoo [43] offers several pre-trained models implementing this network, along which a selection has been carried out (as it will be depicted in other tests). The chosen model integrates a MobileNetv1 whose weights have been quantized [54] in order to reduce the computational cost without reducing the accuracy.

In order to quantify the different accuracy vs. inference time tradeoffs that these architectures offer, a specific test has been designed. A specific video sequence has been recorded containing a person wandering across the field of view of the camera. Several extracted frames from this sequence can be observed on Figure 3.2. For every frame on the sequence, the persons are detected using YOLO and SSD respectively, and the IoU and the inference time have been measured, as it can be seen on Figure 3.3. Some gaps can be noticed on the detections, corresponding to the frames where the person was out of the sight of the camera.



(a) Frame 322. (b) Frame 479. (c) Frame 648.

Fig. 3.2. 3 frames from the test video sequence.

[17] https://pjreddie.com/darknet/yolo/

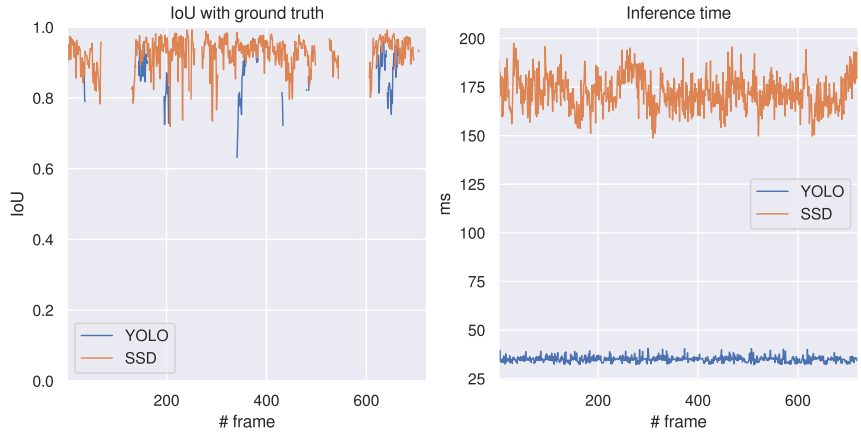


Fig. 3.3. Results of the person detection test: IoU score with ground truth(left) and inference time per frame (right).

3.2. Face detection experiments

One of the improvements of the proposed system over the previous work [11] is the utilization of a fully neural detection pipeline, as it was depicted on chapter 2. This requires the replacement of the face detection Haar cascade classifier explained on section 1.2 by a neural alternative: *faced*.

This experiment is devoted to compare the performance of both face detection systems. Its design is similar to the previous experiment, using the same video sequence (Figure 3.2) with the ground truth faces labeled using LabelMe. For each frame in the sequence, the faces are extracted using each one of the described methods, and the IoU score is computed with the ground truth face bounding box. The result can be visualized in Figure 3.4.

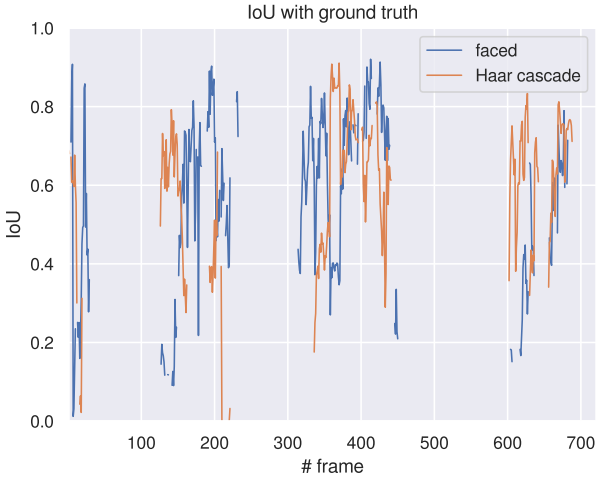


Fig. 3.4. IoU score with the ground truth for each one of the face detection systems.

3.3. Face recognition experiments

The last component of the neural pipeline is a *face recognition* neural network, devoted to confirm the identity of the reference person. This is useful for discerning whether that person has to be followed even if they turns back later, as their position is tracked with the described means. This subsystem is based on a FaceNet [31] network, which projects a face into a 128-dimensional space. These projections are used by the proposed system, as their distance to the projection of a reference face is used to determine if the input face belongs to the reference person.

This experiment is designed to assess the quality of the projection system, which should yield far points for a different face and near points for a matching face. For this proposal, a video sequence was recorded containing two persons wandering in front of the robot. The faces of each frame are labeled, separating the faces of the two persons in two different classes. A caption of the video with the labels can be seen on Figure 3.5. Figure 3.6 shows several frames from the sequence as well, where some occlusions on the faces can be observed.

Fig. 3.5. A frame of the test sequence showing the labels on the faces.

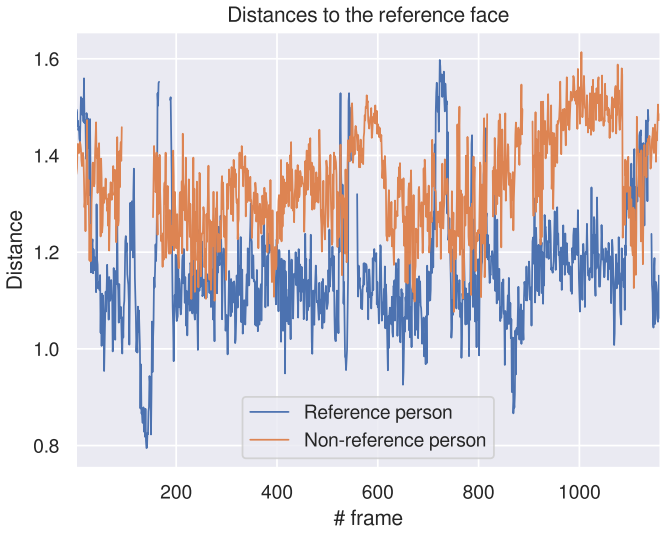
(a) Frame 74. (b) Frame 735. (c) Frame 1136.

Fig. 3.6. 3 frames from the test video sequence.

For computing the distance, the reference face was set using the image on Figure 3.7a,

and the distance to the reference face of each one of the faces in the video was stored. The result can be observed on Figure 3.7b.

(a) Image of the reference face.



(b) Resulting distance for the two faces present in the video.

Fig. 3.7. Results of the face recognition experiments

3.4. TensorRT experiments

3.4.1. Performance tuning the optimization parameters

On section 2.3, the TensorRT engine was introduced. This engine is used to optimize, using a binding component between TensorFlow network graphs and TensorRT itself, the deployment of a neural network on a compatible NVIDIA GPU. There are several tunable parameters for customizing the implementation, and the most relevant ones are described in section 2.3 as well. As varying these parameters changes the model size and the inference time, a benchmark has been developed in order to test the inference time of each model. The optimization script performs a grid search between a set of values for each parameter (MSS, MCE and precision mode, as described on section 2.3), and tests the performance on a specific ROSBag sequence, storing the detections and the inference times on a YAML file, besides the optimized graph to be loaded without requiring to perform the optimization again.

The inference times for the fastest SSD-based model and the Tiny YOLOv3 implementations are shown below in Table 3.1 and Table 3.2. The performance tables for the rest of models can be found in chapter 4.2.

Precision	MSS	MCE	Avg. inference time (ms)
FP32	3	1	59,223
		3	57,139
		5	58,210
	20	1	58,398
		3	58,240
		5	57,910
	50	1	41,077
		3	41,410
		5	41,080
FP16	3	1	57,423
		3	56,777
		5	57,286
	20	1	56,783
		3	56,591
		5	56,637
	50	1	40,053
		3	**39,738**
		5	40,115
INT8	3	1	62,859
		3	61,105
		5	62,383
	20	1	62,439
		3	61,810
		5	63,477
	50	1	46,123
		3	46,835
		5	47,387
GPU without TensorRT			172,269
CPU			112,111

Table 3.1. Grid search results for the `ssd_mobilenet_v1_0.75_depth_coco` model. The lowest inference time is **boldfaced**.

Precision	MSS	MCE	Avg. inference time (ms)
FP32	3	1	20,898
		3	21,032
		5	21,112
	20	1	21,373
		3	21,208
		5	21,639
	50	1	22,506
		3	22,301
		5	22,239
FP16	3	1	16,180
		3	**15,922**
		5	16,061
	20	1	16,200
		3	16,208
		5	16,183
	50	1	18,294
		3	18,110
		5	18,248
INT8	3	1	35,266
		3	36,329
		5	36,289
	20	1	36,305
		3	35,420
		5	35,734
	50	1	35,195
		3	34,815
		5	35,178
GPU without TensorRT			35,996
CPU			NHWC

Table 3.2. Grid search results for the `yolo_v3_tiny` model. The lowest inference time is **boldfaced**. The CPU inferences could not be performed due to hardware incompatibility issues.

3.4.2. Optimized graphs vs. standard graphs

As it has been studied, tuning the TensorRT optimization parameters greatly varies the inference time required for processing an image. However, as it was explained in section 2.3, this acceleration additionally entails a reduction on the precision, as the weights of the neural network layers are trimmed in the process. The precision mode choice determines the precision of the weights. In the case of the SSD-MobileNet detector, the best inference time (Table 3.1) was yielded by the FP16 precision mode, which trims the weights to a 16-bit long floating point number. This will cause the inference precision to be reduced as the operations are performed on a coarser mode.

This experiment aims to quantify the loss of precision when the SSD model is optimized by TensorRT using the FP16 precision model, which is the fastest mode to infer, as shown in Table 3.1. To do so, the test sequence (Figure 3.2) is used again, passing each frame forward on the standard neural network and storing the detected persons. Later, the same video sequence is passed through the TensorRT version of the same graph, storing the detections of each person as well. When both passes are performed, the IoU score is computed on each frame between the standard inferences (considered the ground-truth labels) and the TensorRT inferences. This IoU score on each frame, along with the inference times for each network model, can be seen on Figure 3.8.

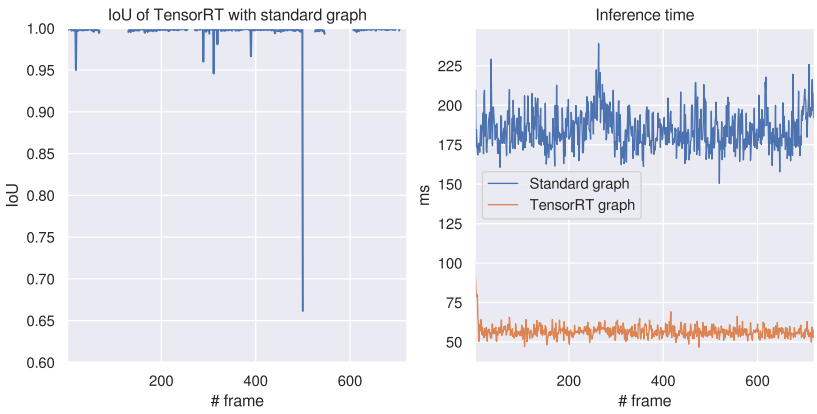


Fig. 3.8. IoU between the standard graph and the TensorRT graph inferences (left) and inference times for both networks (right). The IoU graph has been rescaled between 0.6 and 1 to have a better visualization of the IoU variability.

3.5. Motion tracker experiments

In section 2.3, the Lucas-Kanade tracker was described. This tracker aims to follow the movements of the person between two consecutive inferences from the neural pipeline. As in embedded systems these inferences might take a long time, an interpolation of the detections using optical flow can be crucial for avoiding a loss of the location of the

person, especially if a partial occlusion of the person causes that the network does not detect them for a while.

This experiment aims to identify the conditions under which a tracker can palliate these drawbacks of the neural detection pipeline, depending on the parameter k. This k modulates the number of elapsed frames between two consecutive neural detections, and takes a higher value if the inferences take longer to be computed by the neural pipeline. On the test, a specific test sequence was recorded and labeled, using a hanging blanket with the purpose of partially occlude the person, making the network to lose the detections. Several frames of the sequence can be visualized on Figure 3.9.

(a) Frame 285. (b) Frame 788. (c) Frame 864.

Fig. 3.9. 3 frames from the test video sequence.

A correctly tuned tracker maintains the detection and moves the bounding box for a number of frames (determined by the *patience* parameter, as described in section 2.3). The video sequence was evaluated using $k = 10$ and $k = 20$, checking the influence of the tracker in the IoU with the ground truth labels of the sequence. The result for both values of k can be observed on Figure 3.10, where the lapse corresponding to the person occlusion has been emphasized.

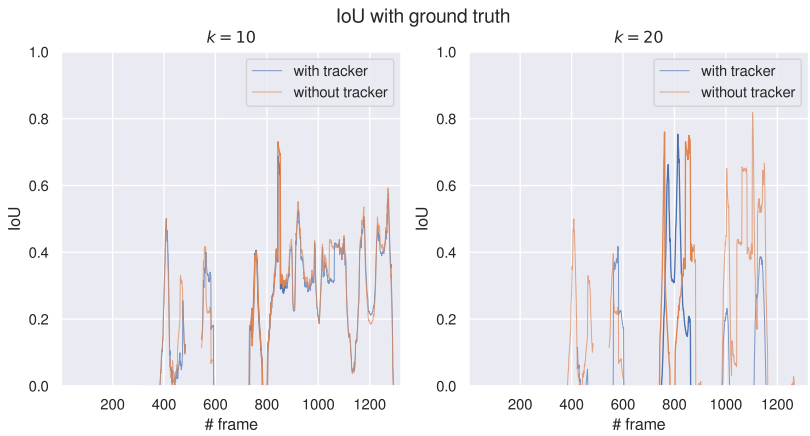
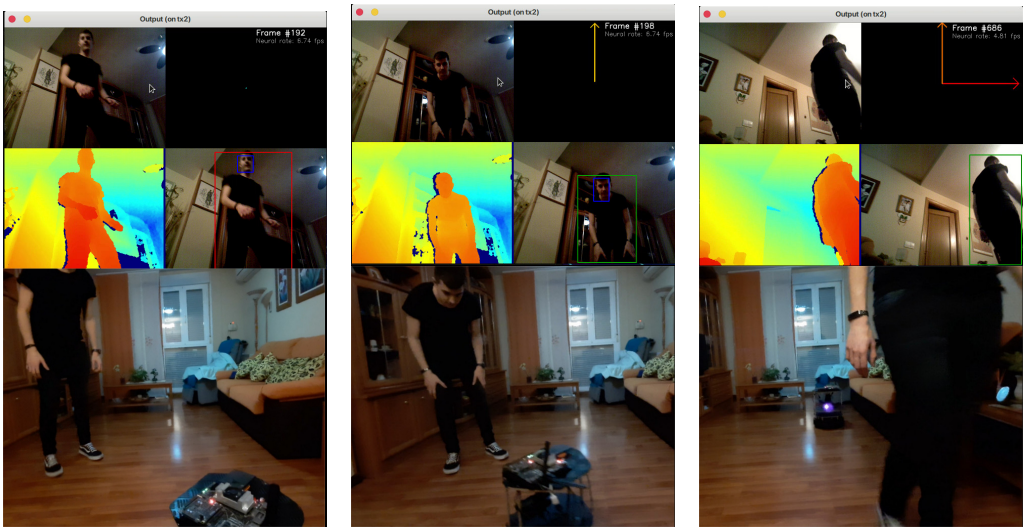


Fig. 3.10. Results of the motion tracker test. The lapse corresponding to the person occlusion has been emphasized.

3.6. Global system experiments

Finally, a visual assessment can be derived from a sample of the fully functional system[18]. As it can be seen on Figure 3.11, the behavior followed by the robot is the expected one.



(a) Person detection. (b) Person recognition and following. (c) Following without face feedback.

Fig. 3.11. 3 frames from the full test (available on YouTube).

[18] https://www.youtube.com/watch?v=WZ0riKMwJWA

4. DISCUSSIONS

On this last chapter, the results obtained in the chapter 3 experiments are analyzed, extracting the conclusions that have led to the final implemented design, that can be seen working on section 3.6. Later, the objectives summarized on section 1.3 are individually revisited, analyzing the degree of accomplishment of the solution on each one of them. Additionally, suggestions are made for further improvements in future works, that can address the drawbacks of the solution proposed in this dissertation.

4.1. Interpretation of the results

The previous chapter described the conducted experiments in order to build the system taking decisions based on objective criterion. On this section, the obtained results will be interpreted, explaining their influence on the choices that have been made for obtaining the final proposal.

4.1.1. Person detection results

On section 3.1, the two outstanding object detection architectures were compared, using both to extract inferences on the same video sequence. The results can be visualized on Figure 3.3. The YOLO-based detector offers a slightly minor IoU than the SSD-based one (around 0.85 and 0.9 respectively), while taking 5 times less time to make inferences (35 ms vs. 175 ms). On these terms, the YOLO-based detector seems much more efficient. However, the IoU graph shows as well a very unstable detection in the YOLO case, being unable to detect the person in most cases. This is shows that this detector is too dependent on pose and lighting conditions for the detections to be successful. On the other hand, the SSD detector yields steady predictions, only cutting on the periods where the person was truly out of the field of view. Hence, this system is much more robust for our application scenario.

One fundamental requirement of the system is the real-time behavior, which makes inference time an important factor to be taken into account. However, as the system includes the described optical tracker, the YOLO detector can be discarded in favor of the SSD-based one, given that the YOLO version has a much lower detection rate[19] and this can not be palliated by the motion tracker.

[19] As it was described before, the deployed version of the YOLO detector is *Tiny YOLOv3*, due to the memory requirements for deploying the full YOLOv3 model, which are higher than what the Jetson TX2 can handle. Thus, it is probable to expect a better performance on the full model in a different computer capable of handling it.

4.1.2. Face detection results

Another important component in the neural pipeline is the face detection network. It has been tested in the experiment conducted in section 3.2, where it has been compared with the face detection tool used on the previous version of the system [11], a Haar cascade classifier [12]. Figure 3.4 shows the detection scores for the two mentioned systems on the same video sequence. It can be seen that both obtain similar IoU scores and drop at the same time when the person turns their back to the camera. However, the `faced` implementation (which uses deep learning to predict the face positions) is capable of keeping a non-zero IoU at several instants where the Haar performance drops to zero. This is due to pose variances of the person, as the main drawback of the Haar cascade classifier is that it is only capable of detecting frontal faces, dropping the performance whenever the person turns the face towards a side.

Hence, this test validates the improvement of the face detection performance when using a specific neural network trained for that purpose.

4.1.3. Face recognition results

The last component of the neural pipeline, the FaceNet face encoder, was tested as well on section 3.3, where the euclidean distances of the projections of two faces were compared to the projection obtained from an external picture of one of them, the *reference face* (Figure 3.7a). The results obtained on Figure 3.7b allows to extract two conclusions about the quality of the projections of the faces:

- The encodings of the reference person (the person with the same face than the reference one) have an overall remarkable stability. In average, the obtained projections for every frame are located at an approximate distance of 1.1 (threshold chosen for accepting a person as the reference one). Exceptional rises in the distance can be found as well, but they are due to changes in the pose of the face, that reduce the quality of the projection.

- The encodings of a person different than the reference one have an overall higher distance from the reference face. This is convenient for avoiding false positives while determining that a face is the reference one.

This allows to conclude a correct performance of the triplet loss (Figure 1.21) on which a FaceNet is trained [31]. This yields an efficient separation between the encodings of different persons, as well as close encodings for faces belonging to the same person,

making this system a robust approach to perform person recognition tasks.

4.1.4. TensorRT results

Once the neural pipeline is established and validated, the TensorRT optimizations can be tested. On subsection 3.4.2, Figure 3.8 illustrates the improvement between a standard graph and an optimized one. One of the premises of the optimization process is the reduction of the precision of the parameters in the neural network, which can be reduced from 64-bit values up to 16-bit or even 8-bit (performing an additional quantization process), as it was depicted on section 2.3.

The loss of precision is patent as well on Figure 3.8, as the IoU between the standard graph and the optimized graph drops at several frames. However, this loss of precision is practically null, with some observable exceptions with a loss below 5% of the original performance. However, the inference time gap can be observed as well. The difference is more notorious, as the TensorRT optimized model performs the inferences 3 times faster than the original graph.

Given these results, the TensorRT optimizations are a convenient tool to greatly increase the performance of the system, allowing the slower component (the neural pipeline) to experiment an important reduction on the inference time. As a result, the overall performance is greatly improved, receiving reliable neural updates more often.

As it was depicted on subsection 3.4.1, a set of parameters can be tuned when optimizing a graph with TensorRT, yielding different performances. As Table 3.1 and Table 3.2 show, important reductions on the inference time can be obtained when the rest of parameters (*Maximum Cached Engines* and *Minimum Segment Size*) are tuned as well. However, these parameters only affect the inference time, as the precision loss is only due to the *Precision Mode* parameter, which has been already analyzed.

The resulting models can be loaded in the program, instead of the original TensorFlow graphs, and offer an overall high performance.

4.1.5. Motion tracker results

Section 3.5 describes the experiment designed for testing the improvement of the optical tracker used between inferences of the neural pipeline. This tracker aims to interpolate the person detections using optical flow, as well as solving the problem of partial occlusions.

The results on Figure 3.10 show the IoU score between the persons and the ground truth labels on the test sequence. Regardless of the value of k (the number of frames elapsed between neural detections), a similar performance can be appreciated under standard conditions (the faded portion of the graph). However, the emphasized region corresponds to an occlusion behind a hanging blanket (Figure 3.9). On this lapse, a better performance is perceptible, especially when the inference time of the neural pipeline is higher ($k = 20$). The hanging blanket occludes the person, causing the neural network to stop detecting her. However, as the tracker retains the detection for several updates because of the patience parameter, the person is not lost until several frames later. Additionally, the Lucas-Kanade algorithm allows to determine the displacement of the person even when it is not being detected by the neural pipeline. This explains the higher IoU when the tracker is active due to the bounding box shifting computed using Lucas-Kanade, confirming the improvement in the performance when using the tracker.

Additionally, while the neural pipeline runs on the GPU of the board, the Lucas-Kanade tracker, whose calculations are much lighter, runs on the CPU. This separation allows to combine both systems asynchronously without an affection on the overall load of the system.

4.1.6. Global system results

The full system can be seen in action on the recorded video sample available online[20]. This video presents a sequence of the robot following the typical use case: the reference person enters into the field of view of the robot, showing their face to the camera. After some consecutive frames detecting the person, it is identified using the detected face. When its projection is close enough to the reference one, the robot starts following the person. For each frame, the linear and angular errors are computed, even if the face of the person is not seen anymore, as the system has checked previously that the person has to be followed. If the errors are outside the safe zones, a velocity command is computed and sent to the robot. This routine is executed iteratively until the person gets lost, causing the robot to stop waiting for the person to be seen again.

[20] https://www.youtube.com/watch?v=WZ0riKMwJWA

4.2. Conclusions

This section revisits the objectives stated in section 1.3, and reviews the degree of accomplishment of each one of them

Regarding the first objective, this work has been focused on the development and testing of an embedded system that follows a reference person on a robust way, using the robustness of deep learning for being capable of working in real environments. This project has been developed using an accessible educational robot and a consumer RGBD sensor.

As the second objective requires, the detection and recognition pipeline has been exclusively designed using deep neural networks, ensuring a robust performance in non-controlled environments. As it has been seen along the project description, this robustness is crucial, especially because the image source is an image placed in a low height: the lens has an vertical inclination in order to see the full body of the persons in front of the robot. However, this causes as well an excessive amount of light from ceiling lamps to enter into the camera, dimming the persons on the image. As it has been tested in chapter 3, classical systems tend to fail given this issue.

This neural pipeline has been complemented by a tracking component, improving the performance under unexpected issues, such as partial occlusions, or a higher inference time. This could happen if the networks are more complex or the inference device does not provide a low detection time. This fulfills the third objective of the project.

However, further improvements can be done on future works, such as:

- Implement a multimodal tracking using sensor fusion, like in works such as [55]. The depth data of the person also provides information about their position, and bringing this information into the tracker can potentially lead to a better performance.
- Implement a probabilistic tracker, such as an EKF (*Extended Kalman Filter*), taking leverage of the person trajectory. This can avoid confusions between two persons if they cross each other, or help the system to follow the trajectory of a person even if it is temporary lost. In addition, this can solve problems that come up from using optical flow, such as a person moving a part of their body. The displacement of the keypoints on that part of the body cause the full bounding box to suffer a displacement even if the person has not changed its position. This can be addressed using probabilistic subsystems to predict the movement of the person.
- Add a navigation component to the robot. The used robot is additionally equipped

with a laser scanner, it can be used to detect possible obstacles between the robot and the person. Thus, a simple planning algorithm such as VFF (*Virtual Force Field*) can be combined with this system in order to avoid collisions while the robot is moving.

BIBLIOGRAPHY

[1] *Computer Vision Market to Reach $ 48.6 Billion by 2022*, https://bitrefine.group/11-blog/120-establishing-your-brand-on-college-campuses, Accessed: 2020-06-07.

[2] A. Radford *et al.*, "Language models are unsupervised multitask learners," 2019.

[3] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8599–8603.

[4] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[5] P. Martínez-Olmos, "Deep Learning course: Convolutional Neural Networks," University Lecture, 2020.

[6] J. Potel, "Trial by Fire: Teleoperated Robot Targets Chernobyl," *IEEE Computer Graphics and Applications*, 1998. [Online]. Available: https://www.computer.org/csdl/mags/cg/1998/04/mcg1998040010.pdf.

[7] P. Berkelman and J. Ma, "The university of hawaii teleoperated robotic surgery system," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2007, pp. 2565–2566. DOI: 10.1109/IROS.2007.4399550.

[8] A. M. Okamura, "Methods for haptic feedback in teleoperated robot-assisted surgery," *Ind Rob*, vol. 31, no. 6, pp. 499–508, Dec. 2004, 16429611[pmid]. DOI: 10.1108/01439910410566362. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1317565/.

[9] D. Girimonte and D. Izzo, "Artificial intelligence for space applications," in *Intelligent Computing Everywhere*, A. J. Schuster, Ed. London: Springer London, 2007, pp. 235–253. DOI: 10.1007/978-1-84628-943-9_12. [Online]. Available: https://doi.org/10.1007/978-1-84628-943-9_12.

[10] R. Gockley, J. Forlizzi, and R. Simmons, "Natural person-following behavior for social robots," in *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '07, Arlington, Virginia, USA: Association for Computing Machinery, 2007, pp. 17–24. DOI: 10.1145/1228716.1228720. [Online]. Available: https://doi.org/10.1145/1228716.1228720.

[11] I. Condés and J. Cañas, "Person Following Robot Behavior Using Deep Learning: Proceedings of the 19th International Workshop of Physical Agents (WAF 2018), November 22-23, 2018, Madrid, Spain," in. Jan. 2019, pp. 147–161. DOI: 10.1007/978-3-319-99885-5_11.

[12] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," vol. 1, Feb. 2001, pp. I–511. DOI: 10.1109/CVPR.2001.990517.

[13] I. González-Díaz, "Computer Vision: Image classification," University Lecture, 2020.

[14] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, 886–893 vol. 1.

[15] I. González-Díaz, "Computer Vision: Local Invariant Features," University Lecture, 2020.

[16] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, 2013. arXiv: 1311.2524 [cs.CV].

[17] R. Girshick, *Fast R-CNN*, 2015. arXiv: 1504.08083 [cs.CV].

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *Lecture Notes in Computer Science*, pp. 346–361, 2014. DOI: 10.1007/978-3-319-10578-9_23. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10578-9_23.

[19] W. Liu *et al.*, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, pp. 21–37, 2016. DOI: 10.1007/978-3-319-46448-0_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[20] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2014. arXiv: 1409.1556 [cs.CV].

[21] A. G. Howard *et al.*, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV].

[22] J. Hosang, R. Benenson, and B. Schiele, *Learning non-maximum suppression*, 2017. arXiv: 1705.02950 [cs.CV].

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*, 2015. arXiv: 1506.02640 [cs.CV].

[24] J. Redmon and A. Farhadi, *Yolo9000: Better, faster, stronger*, 2016. arXiv: 1612.08242 [cs.CV].

[25] ——, *YOLOv3: An Incremental Improvement*, 2018. arXiv: 1804.02767 [cs.CV].

[26] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[27] P. Li, H. Wu, and Q. Chen, "Color distinctiveness feature for person identification without face information," *Procedia Computer Science*, vol. 60, pp. 1809–1816, Dec. 2015. DOI: 10.1016/j.procs.2015.08.291.

[28] A. Bhattacharyya, "On a measure of divergence between two statistical populations defined by their probability distributions," *Bull*, pp. 99–109, 1947.

[29] B. Johnston and P. Chazal, "A review of image-based automatic facial landmark identification techniques," *EURASIP Journal on Image and Video Processing*, vol. 2018, p. 86, Sep. 2018. DOI: 10.1186/s13640-018-0324-4.

[30] R. Gottumukkal and V. Asari, "An improved face recognition technique based on modular pca approach," *Pattern Recognition Letters*, vol. 25, pp. 429–436, Mar. 2004. DOI: 10.1016/j.patrec.2003.11.005.

[31] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015. DOI: 10.1109/cvpr.2015.7298682. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2015.7298682.

[32] C. Szegedy *et al.*, *Going deeper with convolutions*, 2014. arXiv: 1409.4842 [cs.CV].

[33] K. Q. Weinberger, J. Blitzer, and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," in *In NIPS*, MIT Press, 2006.

[34] I. Itzcovich, *faced: CPU Real Time face detection using Deep Learning*, towardsdatascience.com, Ed., [Online; consulted 9-June-2020], Sep. 2018. [Online]. Available: https://towardsdatascience.com/faced-cpu-real-time-face-detection-using-deep-learning-1488681c1602.

[35] J. Vega and J. Cañas, *PiBot: An Open Low-Cost Robotic Platform With Camera for STEM Education*, Oct. 2018. DOI: 10.20944/preprints201810.0372.v1.

[36] M. J. Islam, J. Hong, and J. Sattar, "Person-following by autonomous robots: A categorical overview," *The International Journal of Robotics Research*, vol. 38, no. 14, pp. 1581–1618, 2019. DOI: 10.1177/0278364919881683. eprint: https://doi.org/10.1177/0278364919881683. [Online]. Available: https://doi.org/10.1177/0278364919881683.

[37] M. J. Islam, M. Fulton, and J. Sattar, *Towards a generic diver-following algorithm: Balancing robustness and efficiency in deep visual detection*, 2018. arXiv: 1809.06849 [cs.RO].

[38] M. Stommel and M. Beetz, "Sampling and clustering of the space of human poses from tracked, skeletonised colour+depth images," Jan. 2013.

[39] I. González-Díaz, "Computer Vision: Image registration," University Lecture, 2020.

[40] Y. Robotics, *Kobuki User Guide*, English, version Version 1.1.0, 24 pp. [Online]. Available: https://docs.google.com/document/d/15k7UBnYY_GPmKzQCjzRGCW-4dIP7zl_R_7tWPLM0zKI/edit, consulted on 2020/06/14.

[41] NVIDIA, *NVIDIA Jetson TX2: datasheet*, English, version Version 1.6, 68 pp. [Online]. Available: https://developer.download.nvidia.com/assets/embedded/secure/jetson/TX2/docs/Jetson_TX2_Series_Module_DataSheet_v1.6.pdf?Q_eTPkb4IeUZi3rN5gB7N0v6ZNPZJwCNZxPvj9Ct8Sc_LlQgmY12RNuTrJ-qovqrtMX6yUoYcSHbAE1mjhZ3FL59_UxlubPypJB7l7doHcbtGLBaGMzSdiT_6TyVOC2H9klPyl0KcEo48G-XtdkdSfBugtRDYMYn1ouZjffwy5NdPfEyyiSe0T5T204iiO2SUQ, consulted on 2020/06/14.

[42] ros.org, *What is ROS?* wiki.ros.org/ROS/Introduction.

[43] TensorFlow, *TensorFlow Object Detection: Model Zoo*, [Online; consulted 28-June-2020]. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

[44] I. González-Díaz, "Computer Vision: Dense Motion Estimation," University Lecture, 2020.

[45] B. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)," vol. 81, Apr. 1981.

[46] OpenCV, *OpenCV: Optical Flow*, [Online; consulted 22-June-2020]. [Online]. Available: https://docs.opencv.org/master/db/d7f/tutorial_js_lucas_kanade.html.

[47] Jianbo Shi and Tomasi, "Good features to track," in *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.

[48] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1999, 1150–1157 vol.2.

[49] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2, 2005, 1508–1515 Vol. 2.

[50] S. El-mashad and A. Shoukry, "Evaluating the robustness of feature correspondence using different feature extractors," Sep. 2014. DOI: 10.1109/MMAR.2014.6957371.

[51] NanoNets, *Introduction to Motion Estimation with Optical Flow*, [Online; consulted 22-June-2020]. [Online]. Available: https://nanonets.com/blog/optical-flow/.

[52] K. J. Åström and R. M. Murray, "Feedback systems: An introduction for scientists and engineers," Tech. Rep., 2004.

[53] K. Wada, *labelme: Image Polygonal Annotation with Python*, https://github.com/wkentaro/labelme, 2016.

[54] G. A. Blog, *Accelerating Training and Inference with the Tensorflow Object Detection API*, [Online; consulted 28-June-2020]. [Online]. Available: https://ai.googleblog.com/2018/07/accelerated-training-and-inference-with.html.

[55] T. Ophoff, K. Van Beeck, and T. Goedemé, "Exploring rgb+depth fusion for real-time object detection," *Sensors*, vol. 19, p. 866, Feb. 2019. DOI: 10.3390/s19040866.

ANNEX

4.3. Optimization results for all the models

The results are shown in a graphical way, as the tables (originally stored in an Excel spreadsheet) contain conditional formatting, showing the inference speed in a color scale.

4.3.1. Object detection models

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
ssd_mobilenet_v1_coco	FP32	3	1	59,473	16,814
			3	59,405	16,834
			5	59,894	16,696
		20	1	58,791	17,010
			3	60,542	16,518
			5	58,944	16,965
		50	1	45,643	21,909
			3	45,730	21,867
			5	45,458	21,998
	FP16	3	1	56,802	17,605
			3	56,895	17,576
			5	57,810	17,298
		20	1	57,616	17,356
			3	56,580	17,674
			5	58,027	17,233
		50	1	43,509	22,984
			3	43,993	22,731
			5	44,194	22,628
	INT8	3	1	66,161	15,115
			3	65,682	15,225
			5	66,154	15,116
		20	1	63,711	15,696
			3	64,674	15,462
			5	64,890	15,411
		50	1	53,062	18,846
			3	52,652	18,993
			5	52,105	19,192
	GPU without TensorRT			163,633	6,111
	CPU			152,126	6,573

Fig. 4.1. Optimization results for the object detection model `ssd_mobilenet_v1_coco`.

4.3.2. Face detection models

These models are the specifically trained for the `faced` library [34]. They have been optimized as well, swapping the originally included models in the package for the TensorRT optimized ones.

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
ssd_mobilenet_v2_coco	FP32	3	1	67,9234	14,722
			3	67,9204	14,723
			5	67,208	14,879
		20	1	68,2606	14,650
			3	67,8827	14,731
			5	67,6093	14,791
		50	1	52,095	19,196
			3	52,0465	19,214
			5	51,9552	19,247
	FP16	3	1	64,6886	15,459
			3	64,2897	15,555
			5	64,7186	15,452
		20	1	64,5814	15,484
			3	64,6789	15,461
			5	63,6143	15,720
		50	1	48,5507	20,597
			3	48,4325	20,647
			5	48,3583	20,679
	INT8	3	1	116,2488	8,602
			3	114,8495	8,707
			5	116,6604	8,572
		20	1	115,1101	8,687
			3	115,4840	8,659
			5	115,0603	8,691
		50	1	101,5242	9,850
			3	101,3347	9,868
			5	100,6825	9,932
	GPU without TensorRT			207,413	4,821
	CPU			194,621	5,138

Fig. 4.2. Optimization results for the object detection model `ssd_mobilenet_v2_coco`.

4.3.3. Face encoding model

This is the FaceNet implementation [31], which has been optimized as well using TensorRT:

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
ssd_mobilenet_v1_0.75_depth_coco	FP32	3	1	59,223	16,885
			3	57,139	17,501
			5	58,210	17,179
		20	1	58,398	17,124
			3	58,240	17,170
			5	57,910	17,268
		50	1	41,077	24,344
			3	41,410	24,149
			5	41,080	24,343
	FP16	3	1	57,423	17,415
			3	56,777	17,613
			5	57,286	17,456
		20	1	56,783	17,611
			3	56,591	17,671
			5	56,637	17,656
		50	1	40,053	24,967
			3	39,738	25,165
			5	40,115	24,929
	INT8	3	1	62,859	15,909
			3	61,105	16,365
			5	62,383	16,030
		20	1	62,439	16,016
			3	61,810	16,179
			5	63,477	15,754
		50	1	46,123	21,681
			3	46,835	21,351
			5	47,387	21,103
	GPU without TensorRT			172,269	5,805
	CPU			112,111	8,920

Fig. 4.3. Optimization results for the object detection model `ssd_mobilenet_v1_0ẏ75_depth_coco`.

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
ssdlite_mobilenet_v2_coco	FP32	3	1	64,864	15,417
			3	63,414	15,770
			5	63,317	15,794
		20	1	64,936	15,400
			3	64,117	15,597
			5	63,980	15,630
		50	1	49,617	20,154
			3	49,107	20,364
			5	49,804	20,079
	FP16	3	1	63,408	15,771
			3	65,282	15,318
			5	64,823	15,427
		20	1	65,206	15,336
			3	63,855	15,661
			5	64,432	15,520
		50	1	49,570	20,174
			3	49,444	20,225
			5	49,243	20,308
	INT8	3	1	77,383	12,923
			3	76,993	12,988
			5	75,952	13,166
		20	1	77,520	12,900
			3	73,522	13,601
			5	76,221	13,120
		50	1	59,531	16,798
			3	59,681	16,756
			5	59,999	16,667
	GPU without TensorRT			164,628	6,074
	CPU			194,621	5,138

Fig. 4.4. Optimization results for the object detection model `ssdlite_mobilenet_v2_coco`.

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
yolov3_tiny	FP32	3	1	20,898	47,853
			3	21,032	47,546
			5	21,112	47,367
		20	1	21,373	46,788
			3	21,208	47,151
			5	21,639	46,213
		50	1	22,506	44,432
			3	22,301	44,841
			5	22,239	44,966
	FP16	3	1	16,180	61,805
			3	15,922	62,808
			5	16,061	62,264
		20	1	16,200	61,730
			3	16,208	61,698
			5	16,183	61,794
		50	1	18,294	54,662
			3	18,110	55,220
			5	18,248	54,799
	INT8	3	1	35,266	28,356
			3	36,329	27,526
			5	36,289	27,556
		20	1	36,305	27,544
			3	35,420	28,232
			5	35,734	27,985
		50	1	35,195	28,413
			3	34,815	28,723
			5	35,178	28,427
	GPU without TensorRT			35,996	27,781
	CPU			NHWC	N/A

Fig. 4.5. Optimization results for the object detection model `yolov3_tiny` (due to hardware compatibility issues, the CPU testing was impossible to perform).

Model	Precision	MSS	MCE	Mean inf. time	Avg, FPS
face_yolo	FP32	3	1	22,889	43,689
			3	22,743	43,970
			5	23,004	43,470
		20	1	49,985	20,006
			3	50,316	19,874
			5	50,671	19,735
		50	1	50,013	19,995
			3	49,767	20,094
			5	50,305	19,879
	FP16	3	1	17,351	57,633
			3	17,257	57,948
			5	17,278	57,878
		20	1	50,108	19,957
			3	50,348	19,862
			5	50,385	19,847
		50	1	50,269	19,893
			3	50,757	19,702
			5	50,324	19,871
	INT8	3	1	50,350	19,861
			3	50,370	19,853
			5	50,116	19,954
		20	1	49,668	20,134
			3	49,823	20,071
			5	50,479	19,810
		50	1	49,682	20,128
			3	49,964	20,014
			5	50,002	19,999
	GPU without TensorRT			50,609	19,759
	CPU			132,850	7,527

Fig. 4.6. Optimization results for the face detector model `face_yolo`.

Model	Precision	MSS	MCE	Mean inf. time	Avg, FPS
face_corrector	FP32	3	1	5,766	173,436
			3	5,590	178,888
			5	5,605	178,403
		20	1	6,000	166,678
			3	5,693	175,670
			5	5,830	171,541
		50	1	5,916	169,042
			3	5,687	175,846
			5	5,855	170,783
	FP16	3	1	5,604	178,460
			3	5,752	173,868
			5	5,603	178,466
		20	1	5,689	175,781
			3	5,620	177,945
			5	5,898	169,563
		50	1	5,795	172,577
			3	6,048	165,355
			5	5,778	173,061
	INT8	3	1	5,974	167,384
			3	5,805	172,268
			5	5,800	172,411
		20	1	5,633	177,513
			3	5,854	170,835
			5	5,974	167,403
		50	1	5,931	168,611
			3	5,846	171,045
			5	5,621	177,904
	GPU without TensorRT			7,863	127,171
	CPU			8,675	115,268

Fig. 4.7. Optimization results for the face corrector mdoel `face_corrector`.

Model	Precision	MSS	MCE	Mean inf. time	Avg. FPS
facenet	FP32	3	1	40,602	24,629
			3	39,490	25,323
			5	40,163	24,899
		20	1	65,552	15,255
			3	66,399	15,061
			5	66,348	15,072
		50	1	63,498	15,749
			3	65,299	15,314
			5	66,123	15,123
	FP16	3	1	44,575	22,434
			3	44,464	22,490
			5	44,889	22,277
		20	1	66,842	14,961
			3	65,593	15,246
			5	65,840	15,188
		50	1	62,697	15,950
			3	66,012	15,149
			5	65,536	15,259
	INT8	3	1	64,552	15,491
			3	NaN	
			5	NaN	
		20	1	66,029	15,145
			3	64,852	15,420
			5	65,507	15,266
		50	1	63,398	15,773
			3	65,662	15,229
			5	64,770	15,439
	GPU without TensorRT			59,562	16,789
	CPU			141,963	7,044

Fig. 4.8. Optimization results for the face encoding model `facenet`.