ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in INGEGNERIA INFORMATICA

TESI DI LAUREA
in
Calcolatori Elettronici T

A deep learning-based approach for 3D people tracking

CANDIDATO:
Matteo Boschini

RELATORE:
Prof. Stefano Mattoccia

CORRELATORE:
Dott. Matteo Poggi

Anno Accademico 2015/16

Sessione I

Table of Contents

1. Introduction 5

2. PeopleTracking 6

2.1. Overview 6

2.2. Stereo Camera and Disparity Maps 6

2.3. Top-view Maps 7

2.4. Tracking Algorithm 10

2.5. Extendibility 12

3. Unmanned Plane Detection 13

3.1. Overview 13

3.2. Operation 13

3.3. Algorithms 17

3.3.1. RANSAC-based Algorithms 18

3.3.2. Hough Transform-based Algorithms 19

3.3.3. Other Algorithms 21

3.4.* Unmanned Plane Detection with *PeopleTracking 21

4. Head Detection 23

4.1. Overview 23

4.2. YOLO 27

4.3. Building the Dataset 29

4.3.1. Data Gathering 29

4.3.2. LHH, LHD and HHD Encodings 30

4.4.* Head Detection with *PeopleTracking 32

5. Experimental Results 35

5.1. Unmanned Plane Detection 35

5.2. Head detection 37

5.2.1. Encodings compared 37

5.2.2. Tracking with Head Detection 43

6. Conclusions and Future Developments 49

7. Appendix: Configuring YOLO 50

7.1. Overview 50

7.2. Downloading Darknet 50

7.3. Acquiring and Labelling images 50

7.4. Preparing the Training Set ... 51
7.4.1. Preparing Files ... 51
7.4.2. Image Encodings ... 53
7.5. Launching the training ... 53
7.6. Testing ... 54
7.6.1. Preparing a Test Set ... 54
7.6.2. Alternative Testing Modes ... 54
7.6.3. Comparing Results ... 55
8. References ... 56

1. Introduction

The purpose of this thesis is the extension of an existing software framework that tracks people moving within the field of view of a 3D camera in real-time. It was developed within the Computer Science and Engineering Department (DISI) of the University of Bologna and makes use of a custom and highly efficient stereoscopic sensor. The possibility of tracking movements offers many possibilities for useful real-world applications including video surveillance, domestic assistance and any kind of data collection concerning human behaviour (e.g., for commercial purposes).

This work aims at enabling an easier set-up of the tracking system and at increasing its reliability.

To pursue this second objective, we took advantage of *deep learning*, which is one of the most discussed and studied branches of modern research in computer science and has become fundamental for many state-of-the-art systems in computer vision.

2. PeopleTracking

2.1. Overview

The technologies developed for this thesis are primarily aimed at improving *PeopleTracking*, a real-time tracking system originally designed by Alessandro Muscoloni [1].

Figure 2.1 sums up the main steps it performs. The information acquired by the stereoscopic camera in the form of **disparity maps** is initially processed in order to separate the **foreground** from the background. The detected foreground is then used to generate the **top-view maps**, a representation of the scene from a point of view that is orthogonal to the plane on which the tracked subjects move. Finally, the **tracking algorithm** yields a description of the subjects' movement by analysing the maps and employing a **Kalman Filter** to predict their future positions.

In this chapter, these elements will be examined in detail.

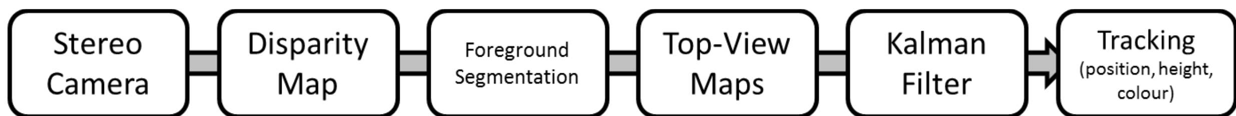


Figure 2.1 - The pipeline of PeopleTracking [1]

2.2. Stereo Camera and Disparity Maps

PeopleTracking relies on frames captured by an RGB-D camera that is capable of providing images containing more reliable information than those acquired by 2D cameras, which are still employed by most tracking systems.

The device has been developed from scratch by researchers at DISI and provides on-board data processing using high-efficiency algorithms mapped on a Field Programmable Gate Array (FPGA). As a result, we obtain a **disparity map**: a picture whose pixels encode depth information. This allows for an accurate scene analysis even

when luminosity varies and permits a precise description of the position of the subjects in space, thanks to the knowledge of the 3D structure of the sensed environment.

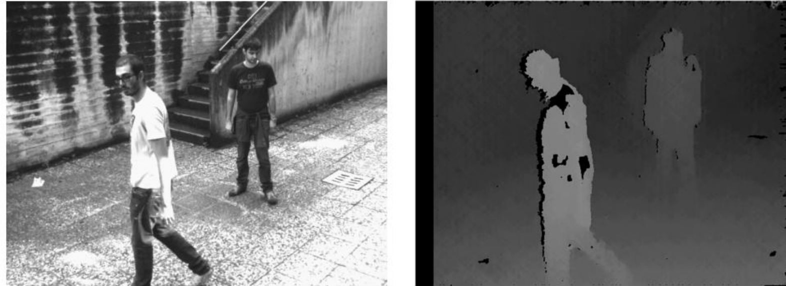


Figure 2.2 - An image captured by the stereo camera and the computed disparity map

At start-up, the program obtains the disparity map of the background that will not be updated at run-time. Every new map that is fed to *PeopleTracking* is initially compared with this one and the pixels whose depth's difference with respect to the background does not exceed a given threshold are discarded.
This simple procedure is extremely fast, since it drastically reduces the amount of data that is handled. Nevertheless, there are some evident limitations to its effectiveness. For instance, any still object entering the scene after start-up would appear as foreground.

Figure 2.3 – An example of foreground segmentation

2.3. Top-view Maps

PeopleTracking has been designed to work on disparity maps captured by a down-looking camera, in a static position, that is placed

slightly above the scene. Due to the high incidence of noise on the disparity maps, the tracking subsystem does not work on them directly, but uses another set of maps instead, that are generated by projecting the 3D points of the foreground on the floor.
The effect of this geometric transformation is a "virtual repositioning" of the camera to an overhead position. Examining the scene from this perspective makes the analysis easier and robust against occlusions and other issues. Virtualising this translation allows us not to place the camera directly above the scene which, for instance, might be challenging when working outdoors due to the need of suitable infrastructures.
It is worth to notice that, in order to make this process possible, the system must be aware of the position of the plane on which the tracked subjects move. This information was gathered by means of an offline procedure by physically placing a checkerboard on the ground. In fact, the patterns can be used by a *calibration program* to determine the plane's equation.

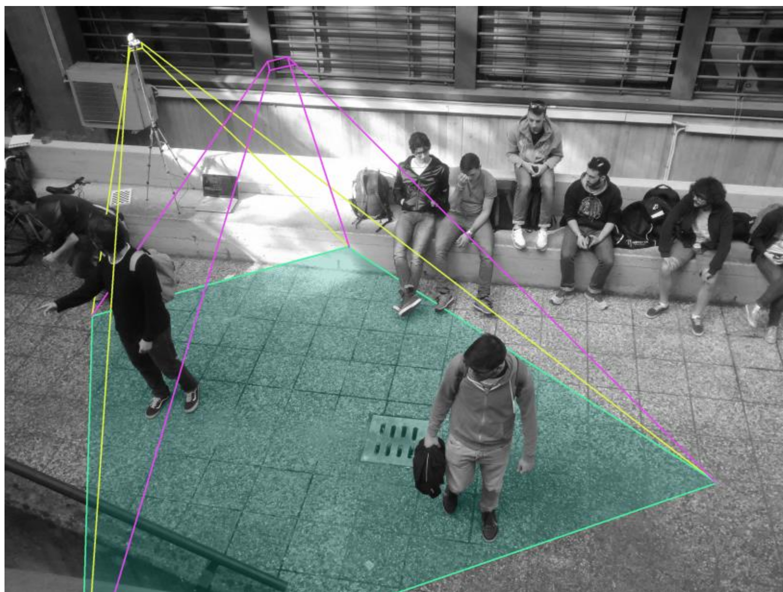


Figure 2.4 - Representation of the virtual repositioning of the camera

So as to generate the **top-view maps**, every point in the point cloud (which is a representation of the scene as a collection of points placed in the three-dimensional space, that can be extracted from

the disparity map) is translated from a coordinate system whose origin is the centre of the camera to a new coordinate system whose x and y axes are parallel and whose z axis is orthogonal to the walking plane. In addition, the space is discretised by dividing it in **bins**: rectangular cuboids with their bases lying on the ground. The maps that will be used in the following steps are then obtained by drawing a pixel for each bin and binding its colour to a specific statistic that we calculate on the points belonging to the corresponding bin.

The statistics that are used to draw the three top-view maps are the following:

- **Occupancy**: the amount of space that is occupied within the bin;
- **Height**: the maximum distance from the ground of a point of the bin;
- **Color**: the average colour of the pixels that belong to the bin.

The occupancy and height maps are the fundamental input for the tracking subsystem, since their analysis allows it to understand if a person is present in the scene. On the other hand, the color map is used to provide a hint for re-identifying people that are being tracked. The maps are finally filtered to reduce the incidence of noise.

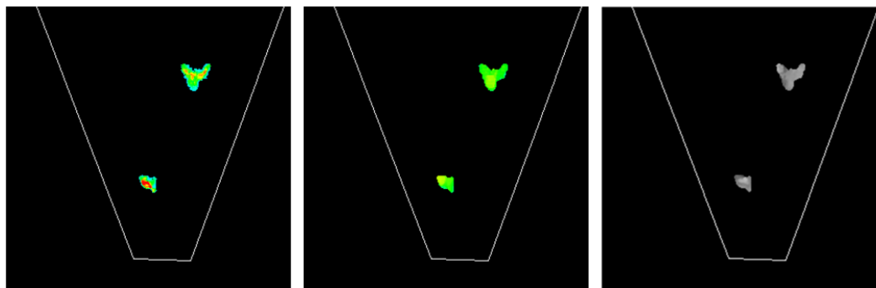


Figure 2.5 - Top-view maps after optimisation

2.4. Tracking Algorithm

The tracking algorithm works by taking the top-view maps as an input and calculates the position of each person on the scene, to whom it associates an unambiguous numeric identifier. It is organised in four steps: *prediction*, *measurement*, *localisation* and *lost subjects' matching*.

The system memorises the following information for each subject:

- their status: a subject can be **in tracking**, **lost** or **candidate for tracking**;
- a Kalman Filter that stores the subject's position and velocity;
- a histogram representing the subject's colour;
- their height;
- for *lost* or *candidate* subjects, the number of frames elapsed since their last status update.

During *prediction*, the position and velocity of every person that is currently being observed are predicted before even examining the new frame.

This is made possible through the use of a **Kalman Filter**, which is a recursive algorithm that can be used to foretell the state of a system depending on its past (condensed in a set of state variables). After each prediction, the filter corrects its parameters depending on how much its guess differs from the actual new state, thus perfecting its accuracy.

The new frame is taken into consideration only in the second phase of the tracking algorithm. During *measurement*, the system searches for every tracked subject in an area that surrounds their predicted position. If the top-view maps show a sufficient value of occupancy and no one else is too close, the person's information is updated and they are erased from the maps. Repeated successes in determin-

ing the position of a *candidate for tracking* trigger their update to the *in tracking* status. A failure in the occupancy test results in the subject being demoted (a *candidate* is simply removed from the system, whereas a subject *in tracking* becomes *lost*).
Any measure on people that are too close to others (and, thus, prone to ambiguity) is suspended and postponed. At the end of this phase, non-critical and measurements are complete and the suspended ones are resolved as well.

As *localisation* begins, the maps only contain information on people that have not been matched with the subjects that were already in the system. Any group of pixels that is high enough and associated with a sufficient occupancy is therefore registered as a candidate.

To complete tracking, an attempt is made to match every lost subject with the candidates by comparing the compatibility of their position, height and colour. If they are found to be similar enough, the candidate is recognised as the lost person and becomes an *in tracking* subject.
On the other hand, if a lost subject is not found to be compatible with anyone currently on scene and their frame count exceeds a specific threshold, the lost subject is removed from the system. This is necessary, since the esteem of their location, which is based on their last recorded position and velocity, becomes less reliable with every new frame.

Figure 2.6 - Tracked subjects shown on the top-view maps

2.5. Extendibility

PeopleTracking has proved to be a fast and reliable system. Whilst depending on a camera positioning that is easily achievable both indoors and outdoors, the requirement of an offline calibration to recognise the ground makes the system’s set-up harder and not up-dating the background used in foreground segmentation might make it unfitting for real-world scenarios where a scene is typically changing.

These are the two issues the we address in this thesis, by intro-ducing a fast unmanned plane detection procedure that can be run online at any time and by replacing the foreground segmentation module of *PeopleTracking* with an advanced recognition system that is based on machine learning.

3. Unmanned Plane Detection

3.1. Overview

As we anticipated in section 3.2, *PeopleTracking* requires an accurate calibration to take place by putting a chessboard or another geometric pattern on the ground, in order to detect the plane on which the tracked subjects move. The object of this chapter is examining how *lib_plane_detection*, a library which was originally developed by Valerio Poli [2] and that I analysed and re-engineered while working at DISI, allows for a simplification of this process and removes human intervention.

3.2. Operation

The library elaborates images that have been acquired from the stereoscopic camera, which contain three-dimensional information in form of disparity maps. Firstly, the maps are used to obtain a point cloud representing the scene. This means that every pixel of the disparity map is associated with its position in space in the "native" coordinate system of the camera (i.e. a system whose z axis comes out of the device towards the scene, as shown in *figure 3.1*).

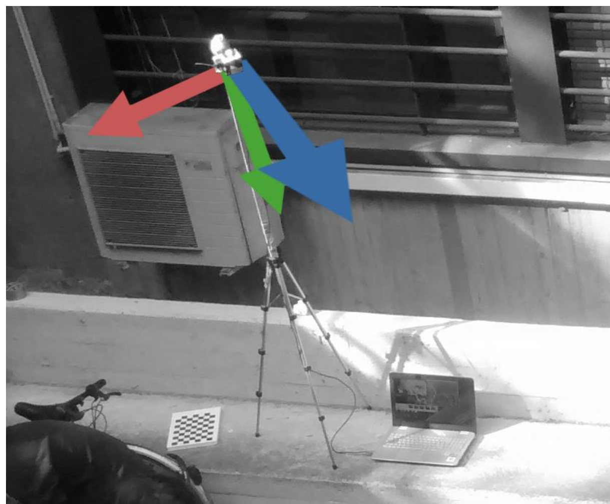


Figure 3.1 - Representation of the "native" coordinate system of the stereoscopic camera

The point cloud extraction is empowered by another library developed within DISI, which is called *lib_pointcloud*.

The plane detection is carried out by using one of the many algorithms that have been included in the library and that are detailed in *section 3.3*. Each one of them analyses the point cloud, recognises the most extensive plane and yields it in its geometric Cartesian equation ($ax + by + cz + d = 0$), which is still relative to the coordinate system mentioned above.

Upon completing the detection, the library establishes a new coordinate system with its x and y axis lying on the plane, which will substitute the one obtained when calibrating *PeopleTracking*. The origin is determined by intersecting the "native" z axis with the plane. Given the equation that represents the plane and the one describing the axis, the coordinates of the point are simply obtained via the following calculations. The result is illustrated in *figure* 3.2.

$$\begin{cases} ax + by + cz + d = 0 \\ x = 0 \\ y = 0 \end{cases} \rightarrow O: (0, 0, -\frac{d}{c})$$

Figure 3.2 - The origin of the new coordinate system.

The direction of the x axis for the new coordinate system is then found by intersecting the “native” system’s xz plane ($y = 0$) with the walking plane and by parameterising the resulting Cartesian equation on the variable z:

$$\begin{cases} ax + by + cz + d = 0 \\ y = 0 \\ z = \lambda \end{cases} \rightarrow \begin{cases} x = -\dfrac{d + c\lambda}{a} \\ y = 0 \\ z = \lambda \end{cases} \rightarrow \begin{pmatrix} -\frac{c}{a} \\ 0 \\ 1 \end{pmatrix}$$

If the resulting direction’s verse is opposite to the one of the “native” x axis (that is if $-\frac{c}{a} < 0$), the newly found x axis is reversed. The result is shown in *figure 3.3*.

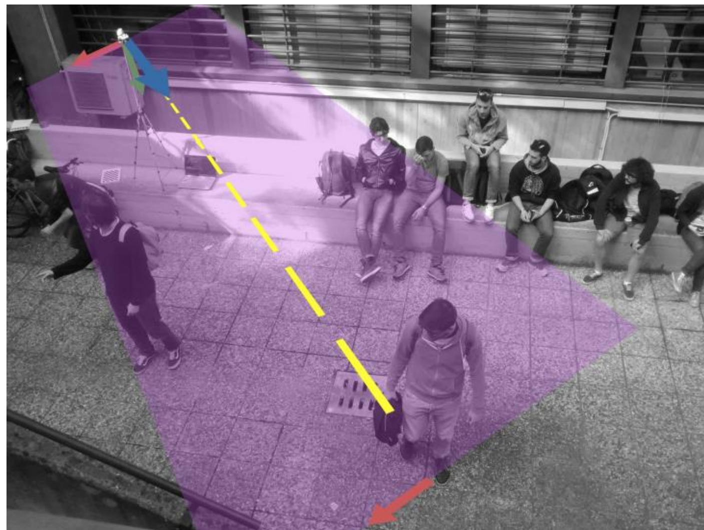


Figure 3.3 – The x axis of the new system if found by intersecting the “native” xz plane with the walking plane.

Similarly, the direction of the y axis for the new coordinate system is found by intersecting the “native” system’s yz plane ($x = 0$) with the walking plane and by parameterising the resulting Cartesian equation on the variable y:

$$\begin{cases} ax + by + cz + d = 0 \\ x = 0 \\ y = \lambda \end{cases} \rightarrow \begin{cases} x = 0 \\ y = \lambda \\ z = -\dfrac{d + b\lambda}{c} \end{cases} \rightarrow \begin{pmatrix} 0 \\ 1 \\ -\dfrac{b}{c} \end{pmatrix}$$

Again, if the resulting direction's verse is opposite to the one of the "native" z axis (that is if $-\frac{b}{c} < 0$), the newly found y axis is reversed. The result is shown in *figure 3.4*.

Figure 3.4 - The y axis of the new system if found by intersecting the "native" yz plane with the walking plane.

Finally, the z axis is simply obtained from the cross product between the other two axes. The resulting new system is shown in *figure 3.5*.

Figure 3.5 – The complete new coordinate system.

The gathered information is then stored in a rototranslation matrix and then saved on the file system for online computations. Let $(x_a, x_b, x_c), (y_a, y_b, y_c), (z_a, z_b, z_c)$ be the directions of the parametric equations of the axes and (O_x, O_y, O_z) be the coordinates of the origin, the matrix will be expressed as follows:

$$R = \begin{bmatrix} x_a & y_a & z_a & O_x \\ x_b & y_b & z_b & O_y \\ x_c & y_c & z_c & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.3. Algorithms

This section contains a list of the plane detection algorithms that are currently available within the library. The ones that belong to the RANSAC and HOUGH families were examined in detail during my work on *lib_plane_detection* and will be thoroughly explained. The remaining algorithms were simply interfaced with the core of the library and will, therefore, only be described briefly. Each algorithm analyses the point cloud in search of the most extended plane in the scene.

3.3.1. RANSAC-based Algorithms

RANSAC-based algorithm were the only ones originally included in the library prior to my re-engineering. I added to the three versions described in [2] (*standard*, *fast* and *LS*) an additional *optimised* variant.

The *standard* RANSAC algorithm works by drawing three random points from the cloud, by calculating the plane that contains them and determining how many points in the cloud belong to it (given a virtual thickness associated to the plane). If the found number of inliers exceeds a specific threshold, the plane is saved as the best guess until another one is found whose inlier count is higher. After a pre-determined quantity of random extractions, the current best guess is returned as the found plane.
In order to speed-up this procedure, it is possible to evaluate whether the number of inliers suffices to exceed the threshold by taking into account a sample of the point cloud. If the test is passed, the total number of points belonging to the plane is calculated on the whole point cloud.

The *fast* version of the algorithm aims at a faster execution and yields the first plane whose inliers exceed the threshold, which must obviously be increased in order to have meaningful results.

The *optimised* variant simply reduces the algorithm's computational cost when run on a sample by only considering the points that were not sampled during the second count, whereas the original algorithm examined the entire cloud.

Lastly, *LS* does not base its evaluation on the number of inliers, but rather on the average distance of the points of the cloud from the plane.

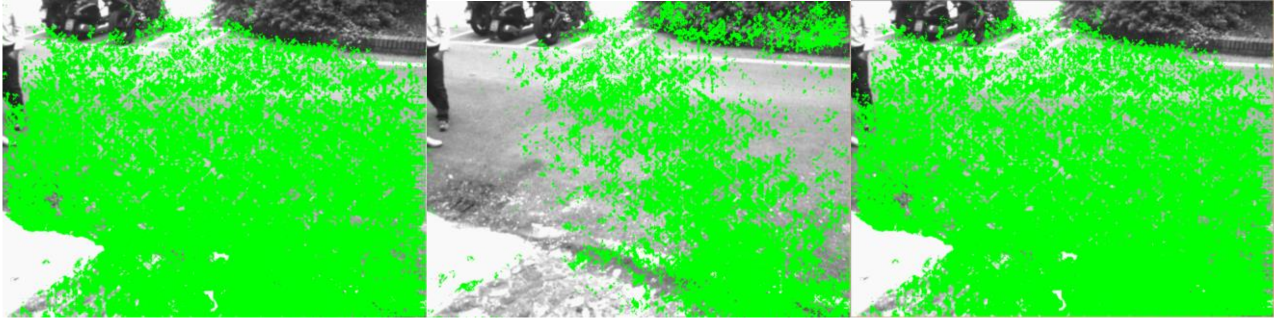


Figure 3.6 – Planes detected with RANSAC algorithms: standard, fast and LS (from left to right)

3.3.2. Hough Transform-based Algorithms

The second class of algorithms was originally developed by Enrico Golfieri in [3] and I interfaced them with *lib_plane_detection*. Due to their use of a non-elementary geometric model, I had to study their working in depth to be able to obtain a Cartesian equation of the output planes.

These algorithms rely on the *Hough transform* technique, which associates every plane in space with the parameters θ, μ and ρ through the following equation:

$$x \sin\mu \cos\theta + y \sin\mu \sin\theta + z \cos\mu = \rho$$

where $\theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right], \mu \in [-\pi, \pi]$. The algorithm considers a finite set of values for θ and μ within their domain, which will be used to discretise them. The same happens for the third parameter, ρ, whose domain is theoretically unbounded, but can be limited to $[-d_{max}, d_{max}]$, where d_{max} is the maximum distance between two points of the cloud. This is possible because a plane's distance from the origin of the coordinate system, using its regular Cartesian equation, is $d_{orig} = \frac{|d|}{\sqrt{a^2+b^2+c^2}}$ and, by substituting a, b, c and d with their corresponding expression in the equation that expresses the *Hough* representation, it is found that $d_{orig} = |\rho|$.

The first algorithm belonging to the class, *Hough Standard*, solves the equation above considering every combination of the possible values of θ and μ for each point in the cloud. Every time the equation is solved, a value for ρ is found. The number of occurrences of a specific triad of parameters is saved in a cell of an accumulator matrix. When the value of one of these cells exceeds a threshold, meaning the plane contains a sufficient number of inliers, it is assumed as the best guess. After every combination of point and parameters has been considered, the best plane is yielded in its Cartesian equation.

The algorithm that was just described is characterised by some relevant issues:

- The discretisation of the parameters can determine significant errors.
- The high number of combinations of parameters that has to be considered implies a long computation time.
- The accumulator that is used to keep track of the number of inliers for every plane may have a major incidence on the computer's memory. In our tests, θ and μ could take 18 values each, whereas ρ could vary within a set of 86100 values and every cell contained a 32-bit unsigned integer. This means that the accumulator, that has to be stored in the computer's RAM, has an overall size of $86100 \times 18 \times 18 \times 4\,Byte \approx 115.9\,MB$. This figure, in case a higher precision is required or the images' resolution is increased, is likely to exceed the typical memory of an ordinary computer.

In order to address these issues, two additional Hough transform-based methods are proposed: the *randomised* Hough algorithms. Both of them adopt a different strategy for choosing what planes have to be considered. Instead of evaluating every possible configuration

of parameters for every point in the cloud, they only examine a reduced quantity of randomly extracted planes, thus decreasing the time required and allowing for a lightweight accumulator.
The first randomised algorithms randomly picks a point from the cloud along with random values for θ and μ. The second method relies on the extraction of three points and on the examination of the only plane in space that contains them (an approach that is very similar to RANSAC).

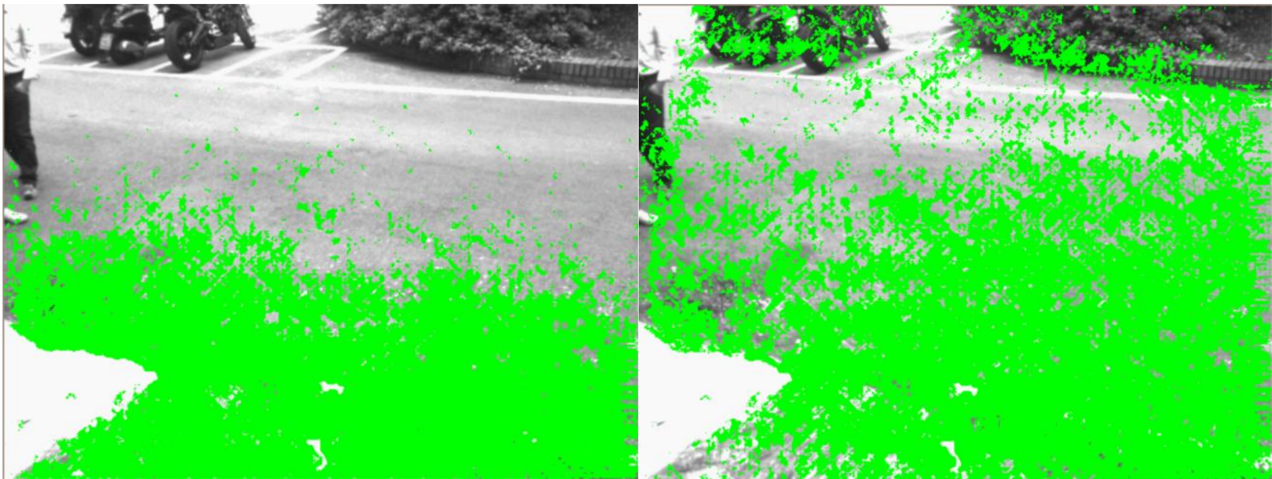


Figure 3.7 – Planes detected with the standard Hough algorithm and its randomised variant (from left to right)

3.3.3. Other Algorithms

Two supplementary algorithms were added to the *lib_plane_detection* as "black boxes" since they only needed minor modifies to have a correct interaction with the library. *Region Growing* was developed by Manuel Rucci [4] and *Normals* was developed by Davide Barchi [5]. The first algorithm revealed some weakness during a meticulous testing carried out by Andrea Garbugli [6] and was subsequently adjusted.

3.4. Unmanned Plane Detection with *PeopleTracking*

Interfacing *lib_plane_detec*tion with PeopleTracking is extremely simple as it only requires launching a plane detection routine be-

fore tracking is initialised. This way, the first frame of the sequence (or a frame captured directly for this purpose if we are working in real time) can be used to obtain an updated rototranslation matrix as described in *section 3.2*, which will be stored on the file system.

During our tests, we discovered that *PeopleTracking* does not work well with the origin we picked above. This issue was solved by choosing an origin O^c as follows:
Be $O: (0, 0, -\frac{d}{c})$ the origin chosen as described in *section 3.2*, and $O': (0, -\frac{d}{b}, 0)$ the projection of the position of the camera on the walking plane along the "native" y axis, the vector that starts in O and ends in O' is $v: (0, -\frac{d}{b}, \frac{d}{c})$. O^C is the point that is obtained by translating the orthogonal projection of the "native" system's origin $(0, 0, 0)$ on the walking plane by $0.7 \cdot v$.

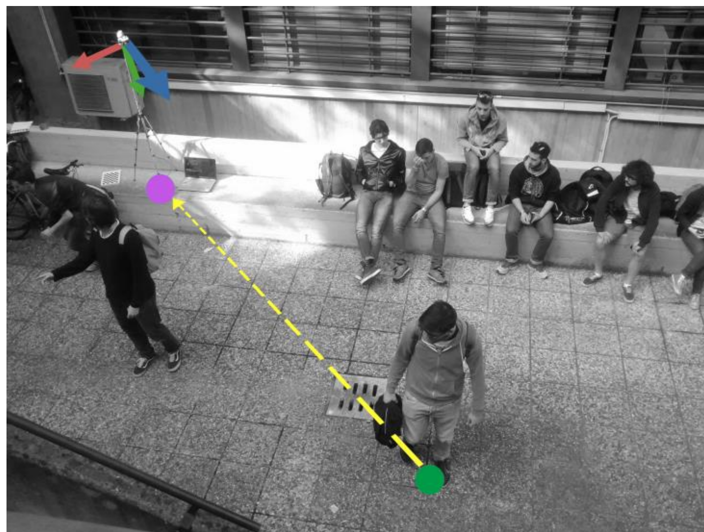


Figure 3.8 - Illustration representing vector v.

4. Head Detection

4.1. Overview

As anticipated in section 2.2, *PeopleTracking* operates a very simple background subtraction for every frame captured by the sensor, in order to separate the scene's foreground. This technique proves to be effective only in very simple scenarios, when the background is made up entirely of still objects. Furthermore, this approach does not help the system distinguishing people from any other object with a similar form factor.
In order to improve the system's functionality and increase its reliability, we decided to add a head detection module to *PeopleTracking*'s pipeline as shown in *figure 4.1*.

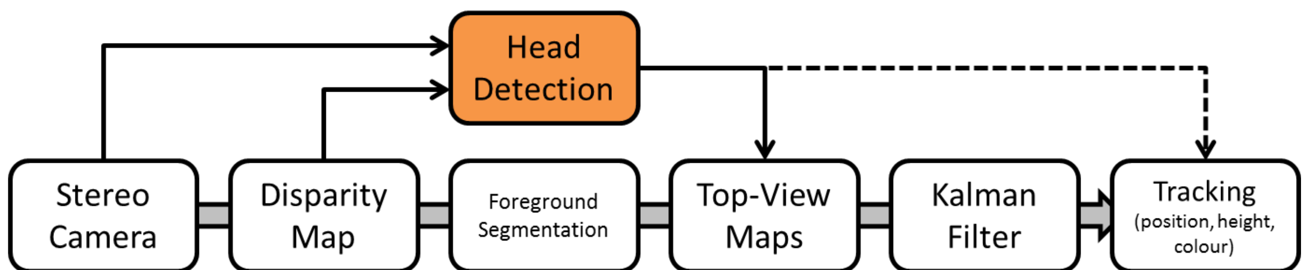


Figure 4.1 – Proposed modification to PeopleTracking's pipeline

Such a module has to be able to analyse the images and disparity maps coming from the sensor and consistently output a set of positions that correspond to people's heads. Human heads have specific features that make their recognition relatively simple and are evidently highly indicative of the presence of a person in their immediate surroundings.

As regards its role in our program, the head detection module would be used to bypass foreground segmentation and could interact with the other parts of the system by **providing a filtering criterion for the top-view maps** or by **establishing every tracked subject's position within the tracking algorithm**.

The first approach is the simplest to implement, as it only requires disabling the foreground segmentation and filtering the top-view maps upon their generation by only keeping track of information for pixels whose distance from the detected head does not exceed a given threshold. This method reduces the information that has to be managed by the tracking module and allows for an increase of the tolerance of its many heuristic parameters, which are used to distinguish people by the size of their body and their minimum and maximum height. These parameters depend on a probabilistic analysis of human features and might be unable to encompass the variety of real-world subjects.

Figure 4.2 - Illustration of the differences between filtering of the scene with background subtraction (left) and with the head detection module (right)

The second approach is more radical than the first one. Reliable information on people's positions entirely removes the need of analysing the top-view occupancy map within the tracking module to determine whether occupancy "spots" are compatible with a person. This would bring about an important simplification of the second and third phases of the algorithm: during measurement we would only need to match the positions that are predicted by the Kalman filter with the ones outputted from the new module and localisation would

only consist of the registration of unmatched subjects as candidates.
Due to time constraints, the current thesis will only describe a system that uses the new module to filter top-view maps, even though a system exploiting the second approach (or even both) would certainly be worth further studies.

With the aim of individuating the most suitable technology for the construction of the head detection module, the state-of-the-art techniques used to address problems of object detection and recognition were examined. The purpose of this research was finding a system that could easily adapt to our scenario and possibly allow real-time operation of the tracking system. In addition, the research was conducted considering the future opportunity to deploy the proposed method on integrated devices (e.g., FPGA + ARM systems such as Xilinx's Zynq). The candidate solutions fall into two categories:

- **Template matching-based approaches**, that are generally fast and simple;
- **Deep learning-based approaches**, are demanding in terms of resources, but typically more reliable

Van Oosterhout et al. applied template matching to stereoscopic images to detect heads and subsequently track people in [7]. In their paper, they capture a sequence from a top-view perspective and compare the images with a spherical shell-shaped template that matches with spheroids whose size is compatible with a human head. This approach is extremely fast and simple, but still relies on the kind of manually tuned heuristic parameters that we are willing to let out of our tracking system.

A fundamental milestone in the study of object detection was reached in 2001 with the proposal of the Viola-Jones object detec-

tion framework in [8], from which derived a family of face detection algorithms that are widely used to this day due to their reliability and efficiency. The standard Viola-Jones algorithm elaborates two-dimensional images in search of simple features that are selected by a machine learning framework and, after a cascade of detection stages, determines whether a human head is present. This solution did not entirely suit our needs, as it is mainly aimed at recognising faces (not heads).

Modern approaches generally rely heavily on deep machine learning. The higher level of abstraction that constitutes the core of this branch of artificial intelligence allows picking features for image recognition with an effectiveness that exceeds all human-crafted templates. However, these systems are much more resource-eager and typically require that the calculations be made on a parallel architecture (such as a modern Graphical Processing Unit (GPU)) to achieve an acceptable speed.
Girshick *et al.*'s proposal of Region-based Convolutional Neural Networks in 2012 [9] has been very influential for the most recent development of computer vision. The proposed system relies on feature analysis of numerous patches extracted from a source image. The patches' extraction ditches the traditional sliding window approach (that requires an *a-priori* knowledge of the aspect ratio of the objects) in favour of a more complex algorithm, called selective search. Even though RCNNs are extremely effective and versatile, they require high-end devices to run at a reasonable speed, which is not entirely appropriate for our goals.
The same can be said for Vu *et al.*'s proposed head detection system [10], that builds up on a dual RCNN to find human heads in frames coming from movies. This system's peculiar evaluation strategy takes into account both "unary features" relative to a single patch where a head might be present and "pairwise features" that examines how couples of possible heads relate to each other to provide fur-

ther validation. While the idea of recognising pairs of heads by studying their interaction is definitely interesting, the high computational cost of using a similar approach makes it incompatible with our application.
A different deep learning-based technique was adopted by J. Redmon *et al.* in [11], where a specialised neural network named *YOLO* (You Only Look Once) is proposed as a lightweight alternative to RCNNs for problems of detection. As its name suggests, *YOLO* only examines the input images once and can work in real-time on modern GPUs, whereas RCNNs typically depend on complex multi-step pipelines that slow them down. This system was chosen as the core of our head detection module because it represents a reasonable compromise between complexity and efficacy.

4.2. YOLO

This section will focus on how *YOLO* works and on what minor modifies were made to make it fit for our purposes.
YOLO is built on *Darknet* [12], an open source neural network framework written by J. Redmon, and operates by unifying what are commonly regarded as separate tasks in a detection problem. Whilst RCNNs typically use region proposal methods to generate potential bounding boxes for the objects that will be detected then post-process the data that is generated to eliminate duplicates and take into account the scene as a whole, *YOLO* scans the full image and predicts bounding boxes for each object in the scene.
The input image is initially divided in a $S \times S$ grid. Each cell is responsible for the detection of up to B objects whose centre falls into its own boundaries. Every time a box is predicted, the network guesses the position of its centre within the image, its width and height and yields a confidence value representing the probability that the box contain an object.

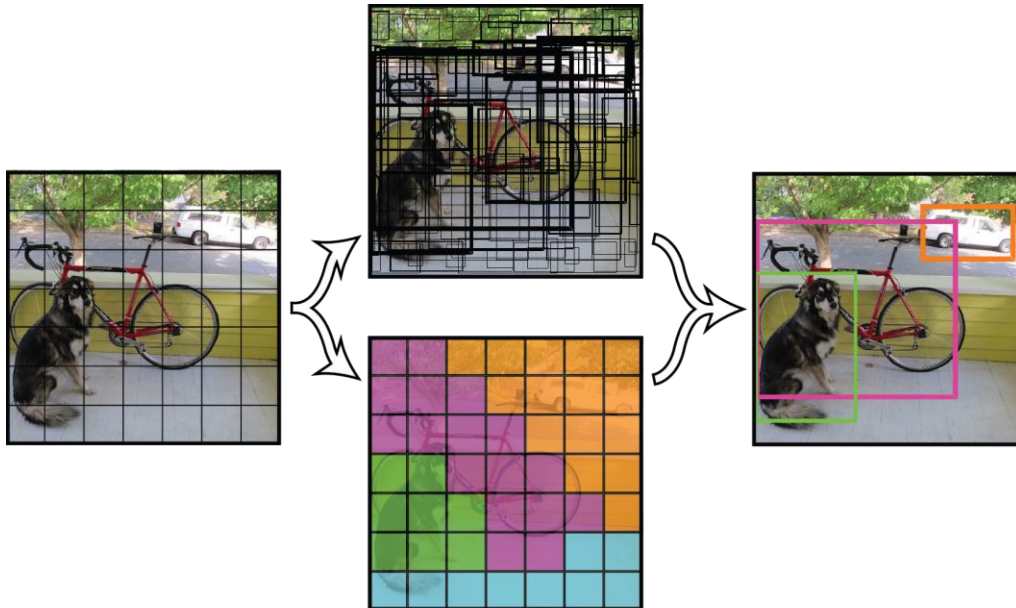


Figure 4.3 – An illustration of YOLO's operation (image taken from [11]). The input image is divided into a $S \times S$ grid. Boxes are then predicted and every cell independently calculates the probability that an of each class be contained in it.

For each of the C classes that the system is trained to detect, every cell also predicts the probability that an object detected inside of it belong to them. These predictions does not depend on the number of boxes B. By multiplying the confidence value of a box with the class probability of its cell we get the probability that the box contain an object of the specified class.
The network architecture is based on *GoogLeNet* model [13] for image classification and consists of 24 convolutional layers, followed by 2 fully connected layers.

The only structural modification that was needed to adapt *YOLO* to our detection task was decreasing the number of classes from 20 (as the network comes configured for usage on the *Pascal VOC 2012* dataset) to 1, that is "head". This simplification is also expected to increase the working speed of the network as well as its reliability.

4.3. Building the Dataset

4.3.1. Data Gathering

In order to train the neural network, a proper dataset needed to be built, made up of images of people and the corresponding position of each head. On 20th and 22nd April 2016 two capture sessions were held in which the stereoscopic camera was used to record scenes with one, two and three people at a time (overall 38871 frames). The work that was made for this thesis only takes into account the first kind of sequences and uses the images from the first session as train set (roughly 10000 valid frames) and the ones from the second session as test set (9000 frames). The two sets are significantly different (see *figure 4.4*) as the images where captured in different locations and involve different subjects.

Figure 4.4 – Two images coming from the dataset we built, respectively from the first and second session.

The acquired images were then carefully labelled, by a team of colleagues, using a simple program, referred to as *HeadLabeller,* developed for this purpose. The labelling process determines the position and size of bounding boxes surrounding heads in a scene for *YOLO* to learn. The output of *HeadLabeller*, a simple comma-separated values-representation of the boxes position and dimension, had to be further elaborated by another utility program, *HefiConverter*,

that generates a set of text files which are natively used by *YOLO* during its training.

Figure 4.5 – Labelled image

The network then needed to undergo a period of training. A full training consists of 40000 iterations of 64 images-batch analysis (roughly 256 epochs). Further details on the procedure for the construction of the dataset and the training of the neural network can be found in *section* 7.

4.3.2. LHH, LHD and HHD Encodings

The typical input for *YOLO*, as shown in [11], consists of standard 3-channel colour images. Since the information captured by the three-dimensional sensor is encoded with single-channel grayscale images, they were initially used by filling with the same information the three channels of RGB images without any further elaboration. The resulting trained system proved to be quite accurate, however additional experiments were made to take advantage of the two spare channels by using different image formats to include additional spatial information.
In [14], Gupta *et al.* describe an alternative image encoding, named *HHA*, for an *RCNN*-based detection system. This encoding uses the channels of an image to convey information about horizontal dispar-

ity, height from ground and the angle between the pixel's local surface normal and inferred gravity respectively. *HHA* is designed to highlight discontinuities in the image, providing the neural network with precious information that it would unlikely learn to compute directly from the disparity map. It was decided to train and evaluate the performance of *YOLO* with three similar encodings.

The first one is called LHH and uses the red channel for the regular monochromatic left image, the blue channel for an image representing the height of each pixel from the walking plane (which is detected via *lib_plane_detection*) and the green channel for the horizontal disparity.

Figure 4.6 - An LHH image and its separate channels: left, horizontal disparity and height.

The second one is called LHD and derives directly from LHH by replacing its green channel with a measure of the density of the cloud surrounding a point. This substitution is justified since in our system *YOLO* is supposed to recognise heads at different distances from the camera. Therefore disparity, which expresses a point's distance in space, is less relevant than density, which can be a useful clue e.g. for distinguishing a head from a hand.

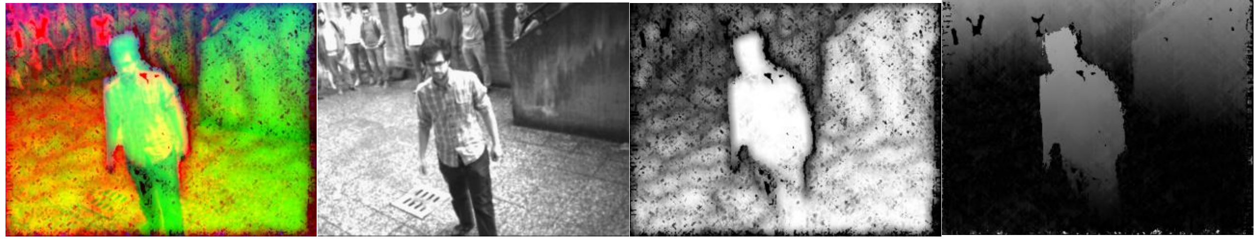


Figure 4.7 – An LHD image and its separate channels: left, density and height.

Finally, HHD uses all the alternative information contained in LHD and LHH and discards the left image. An instance of *YOLO* trained with this encoding was used in synergy with a left-trained version of the neural network in order to try and increase its accuracy.

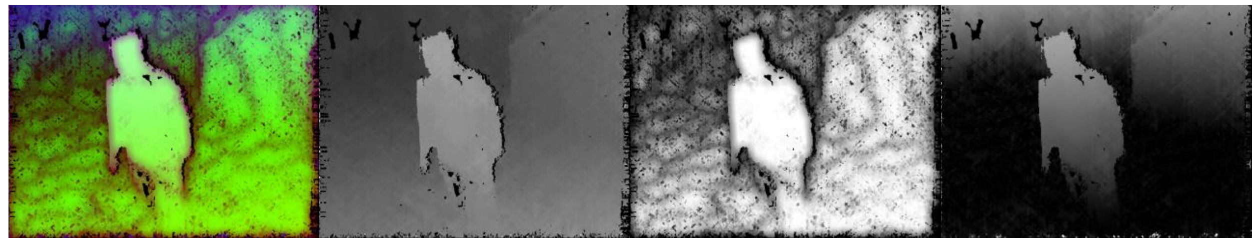


Figure 4.8 – An HHD image and its separate channels: horizontal disparity, density and height.

As shown in *figures 4.6* and *4.7*, LHH and LHD are easily understandable even for a human observer. The different results achieved by *YOLO* using the different encodings are detailed in section 5.2.

4.4. Head Detection with *PeopleTracking*

In order to take advantage of the newly introduced module, as anticipated in section 4.1, the system has to undergo some modifications. During the process that leads to the creation of the top-view maps, background subtraction is disabled and an additional utility binary map is created, called *Head Map*. This map works as a simple mask that filters out all the information that does not refer to bins that are close enough from the estimated position of a head.

Before taking into account the whole disparity map for the current frame, the system remaps the points of the map that correspond to YOLO's predictions. A circular area surrounding the corresponding bins is then marked as valid in the Head Map (in *figure 4.9*, this area is shown in blue). Subsequently, when every pixel of the disparity map is remapped, only those whose corresponding top-view bin lies inside the valid area are used to update the top-view maps.

Figure 4.9 – A frame with its corresponding occupancy map. The points that are outside of the blue area will be filtered out.

So as to enable *PeopleTracking* to read *YOLO*'s prediction, the program has to launch the neural network upon starting up. *YOLO* is executed by a child process that is generated from a *fork* in *PeopleTracking*'s main. The communication between the two resulting processes relies on a *pipe*: *PeopleTracking* sends to *YOLO* the name of the file containing the next image to analyse, then *YOLO* sends back to *PeopleTracking* its predictions, expressed using a simple protocol.

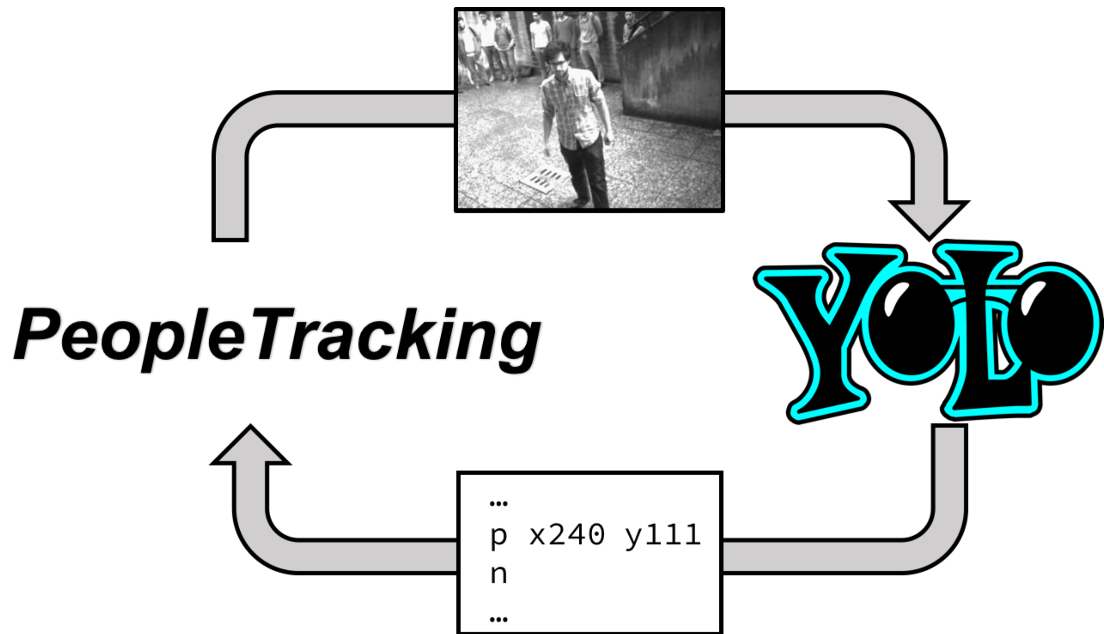
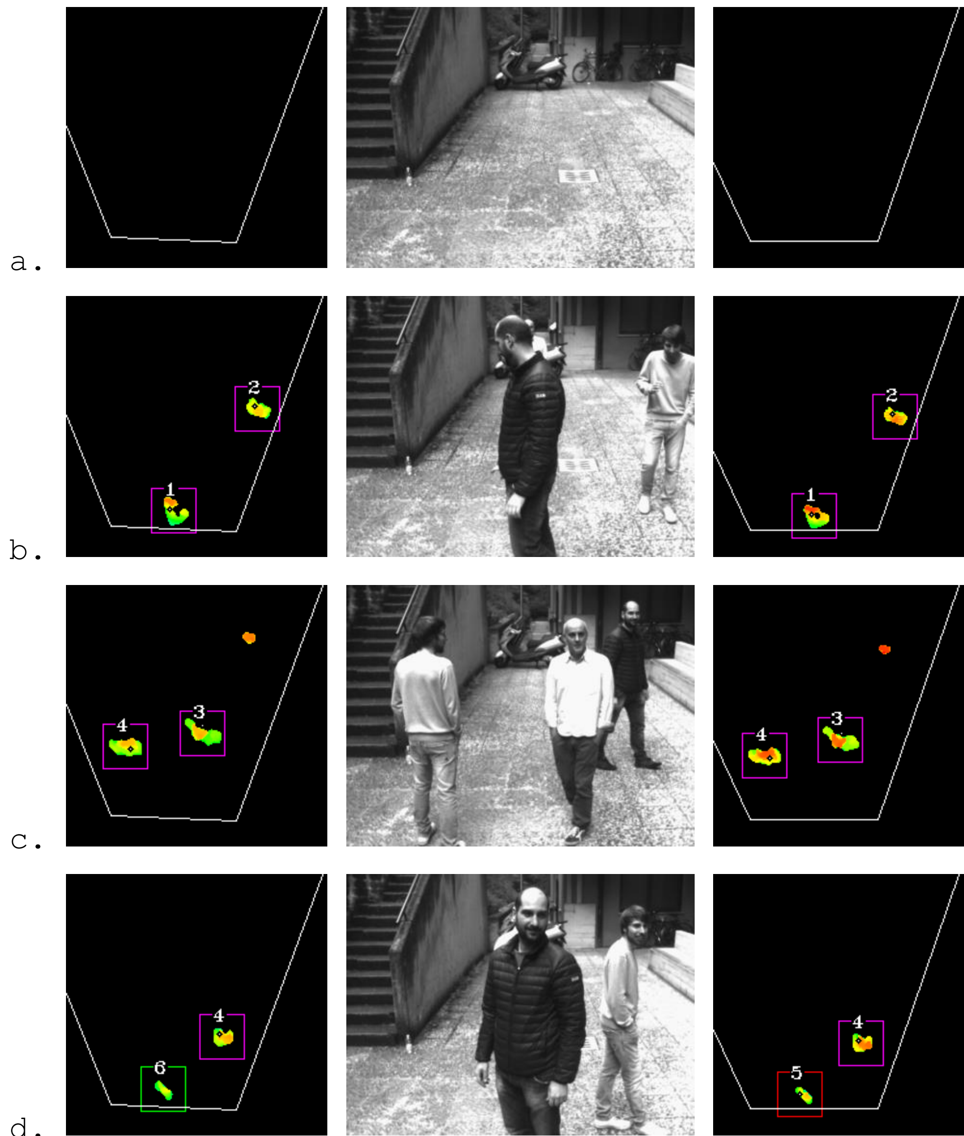


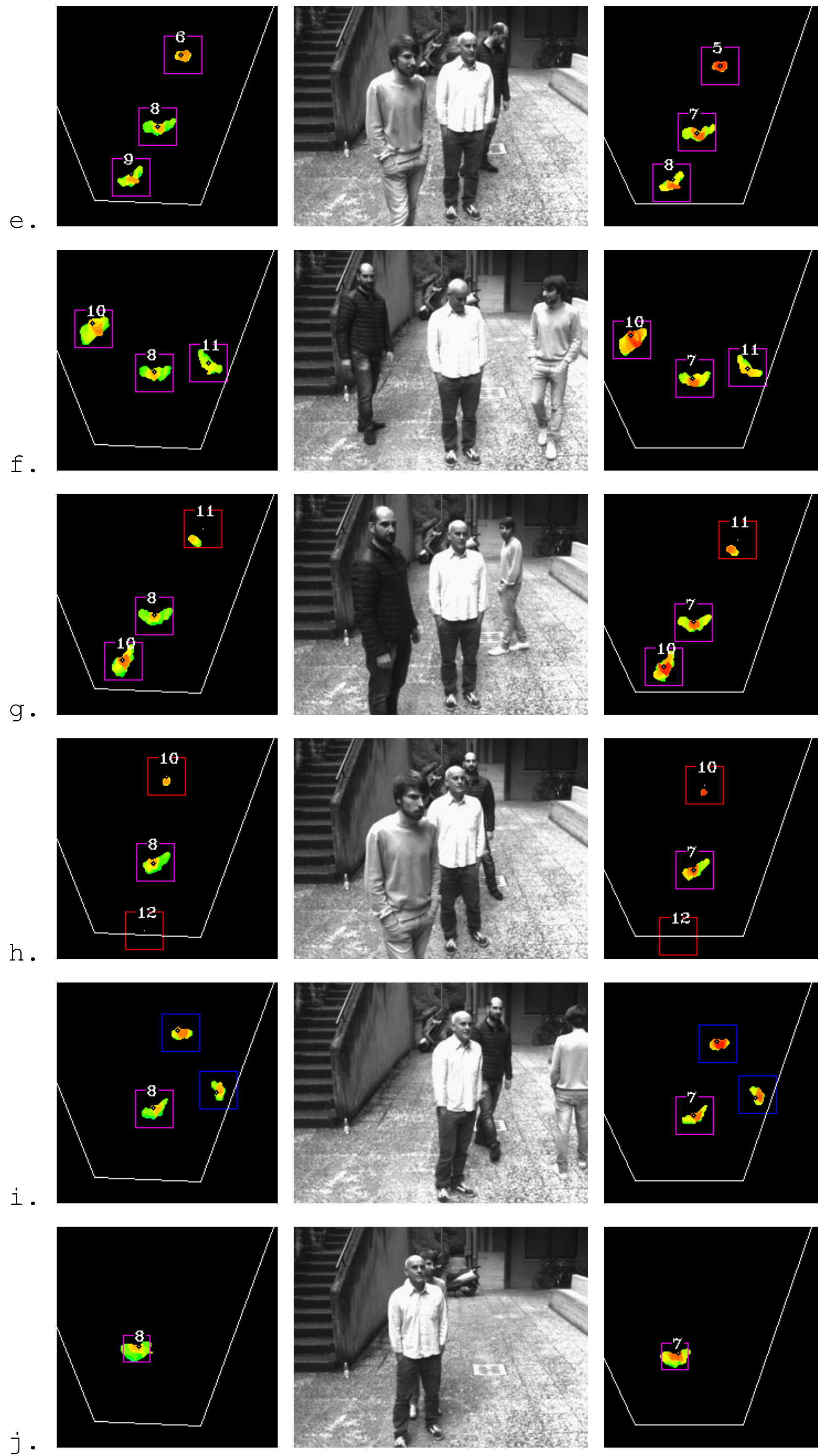
Figure 4.10 - Diagram representing the communication between PeopleTracking and YOLO.

5. Experimental Results

5.1. Unmanned Plane Detection

In *figure 5.1* a comparison is made between the output of *PeopleTracking* when using an offline calibrated rototranslation matrix (left) and when using another one that is obtained from *lib_plane_detection* using the *Region Growing* algorithm (right). The unmanned calibration procedure proves to be effective, which means that an offline calibration is not required anymore.




e.
6
8
9
5
7
8
f.
10
8
11
10
7
11
g.
11
8
10
11
7
10
h.
10
8
12
10
7
12
i.
8
7
j.
8
7

k.

Figure 5.1 – PeopleTracking working with an offline calibrated rototranslation matrix (left) and with the one generated with lib_plane_detection (right).

5.2. Head detection

5.2.1. Encodings compared

As anticipated in section 4.3, *YOLO* was trained four times on a train set consisting of 10094 pictures using different image encodings. After 3000 iterations (roughly 19 epochs), a test was run to determine which training was more effective. The test set includes 6144 images of people walking with plants and bushes on the background.

As shown in the table below, the native grayscale picture encoding proves to be more successful than the "artificial" alternatives. The test revealed an average intersection-over-union between the predicted boxes and the ground truth that is almost three times that of LHH and LHD. This training also proves superior performance in terms of false positives and false negatives reduction.

LHH and LHD show similar stats and prove that *YOLO* is fundamentally confused by their appearance. This might be due to some noise on their channels or to the neural network's being pre-trained on a dataset which makes use of standard pictures (the ImageNet 1000-class competition dataset). The overall superiority of the first training is also confirmed by precision and recall (which are, respectively, measures of how many predicted heads are true positives and of how many labelled heads are predicted) and by different calculations of the F-measure, which is a statistic that combines the

other two with different weights, evaluating the overall performance of the detection.

Encoding	Iterations	Avg. IOU	False Positives	False Negatives	True Positives	True Negatives	Precision	Recall	F-measure		
									$\beta = 0.5$	$\beta = 1$	$\beta = 2$
LEFT	3000	*0.261684*	*21%*	*6%*	*47%*	*25%*	0.687217	0.879915	0.718696	0.771719	0.833189
LHH	3000	*0.089905*	*28%*	*15%*	*38%*	*18%*	0.572990	0.718988	0.597245	0.637740	0.684125
LHD	3000	*0.089888*	*30%*	*12%*	*42%*	*16%*	0.574191	0.778421	0.605989	0.660888	0.726724
LHHD	3000	*0.164053*	*8%*	*28%*	*25%*	*39%*	0.759576	0.705882	0.712133	0.709776	0.707435

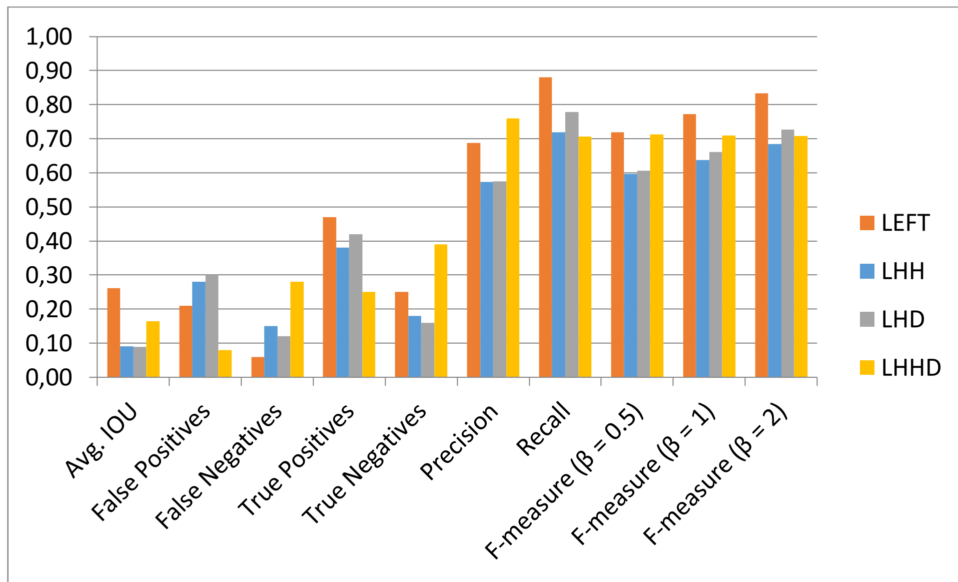
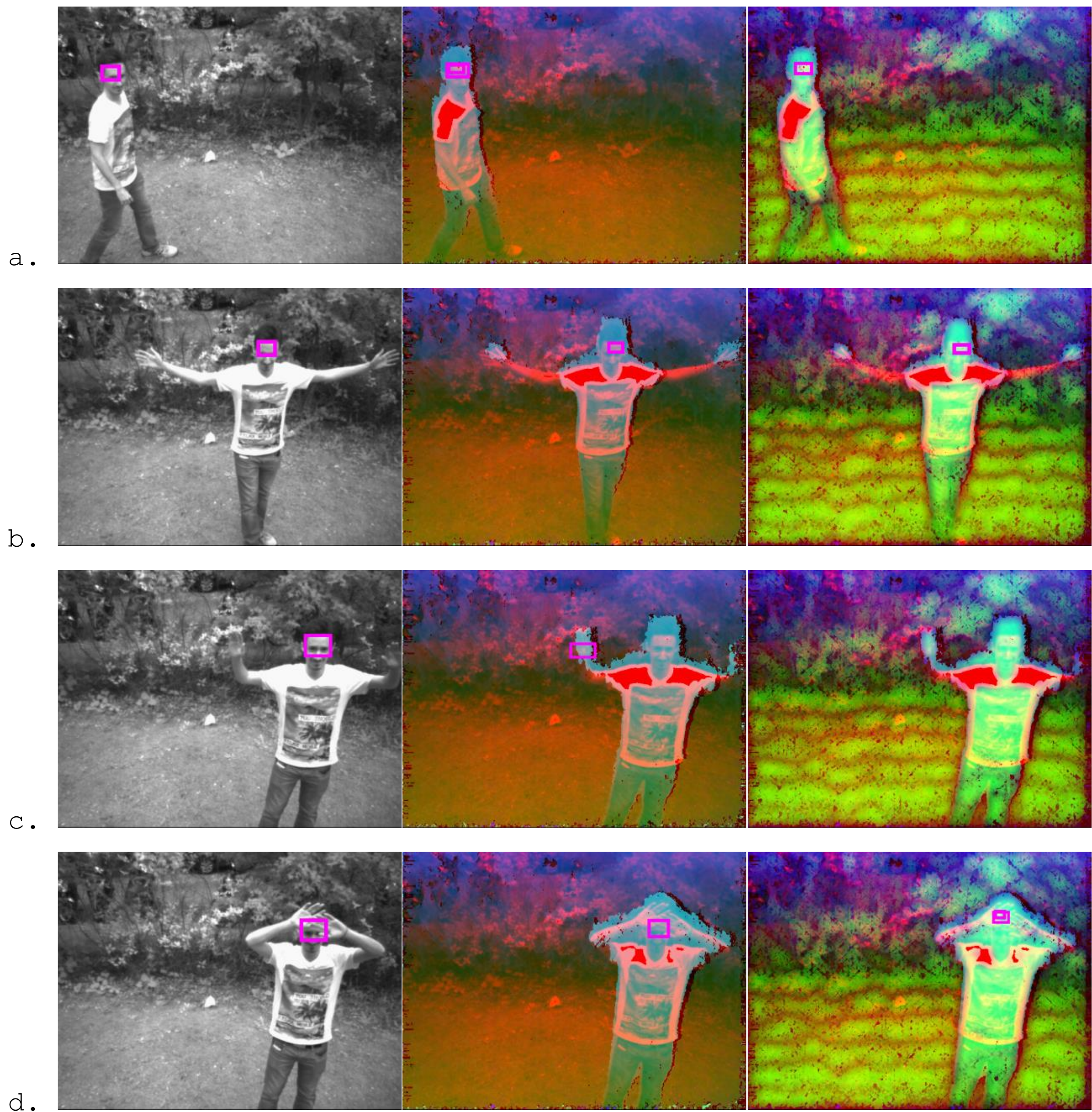


Table 5.1 – (Top) Statistics for the different trainings of YOLO after 3000 iterations. (Bottom) The results are also illustrated in a graph.

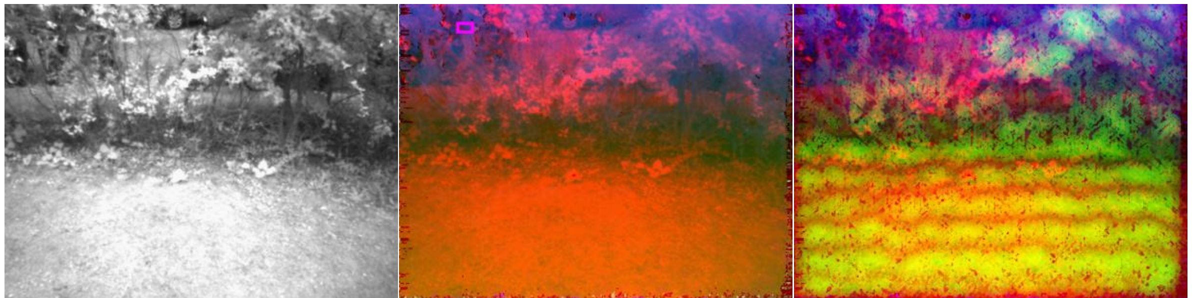
As shown in *figure 5.2*, the left-trained variant of *YOLO* is much less prone to errors and especially to false positives (see frame *g*). Frames *c* and *e* clearly show that, when trained on LHH or LHD, the network is not always able to distinguish a hand from a head (even though the additional information available was supposed to make this distinction easier). Frames *j* and *k* contain a subject that is very close to the camera: since the training set does not contain similar footage, the network is not able to recognise heads in this position. Even so, the left-trained *YOLO* does predict nothing, whereas the other variant predict wrong positions.

Summarising, none of the current trainings allows a precise box sizing (the boxes used in training completely contain the subjects' heads). Nevertheless, in the model of interaction described in *section 4.4*, *PeopleTracking* only depends on *YOLO* for detecting the position of a head. Therefore, it is sufficient that the prediction boxes be centred on actual heads and the results obtained with the left-only training are satisfactory.



e.

f.

g.


h.

i.

Figure 5.2 – YOLO's predictions for some frames of the test set (from left to right, using LEFT, LHH, LHD).

Table 1 also contains the results of the test for a configuration that is referred to as LHHD, which makes use of the left-trained network's predicted boxes but invalidates them if they do not intersect any box coming from an HHD-trained neural network's predictions for the same frame. By combining these data, a consistent de-

crease of the amount of false positives is achieved, although a critical increase of false negatives is also observed.
Due to the model of interaction with *PeopleTracking* that was chosen in *section 4.4*, a high incidence of false positives does not penalise the tracking system as much as an abundance of false negatives. In the first case, a set of points that do not belong to a person will appear on the top-view maps, but they will be typically filtered out by the tracking algorithm if they are not compatible with a human head. Conversely, missing a head would prevent the system from detecting a person.
Therefore, LHHD does not prove to be superior to the other configurations.

After 10000 iterations (roughly 64 epochs), the comparison among the different methods was repeated. Its results generally confirmed the trends that were observed in the previous test and the left-trained neural network still seems to be more effective than the other two variants and the LHHD configuration. As can be observed in *Table 5.2*, progressing with the training makes the accuracy (described by the F-measures) increase, although the false positives increase.
As explained above, this does not compromise the system’s efficacy, while the overall decrease of the amount of false negatives contributes to improving its reliability.

Encoding	Iterations	Avg. IOU	False Positives	False Negatives	True Positives	True Negatives	Precision	Recall	F-measure		
									$\beta = 0.5$	$\beta = 1$	$\beta = 2$
LEFT	10000	*0.333476*	*23%*	*4%*	*50%*	*23%*	0.679982	0.928680	0.718463	0.785107	0.833189
LHH	10000	*0.100266*	*32%*	*12%*	*41%*	*15%*	0.566696	0.776897	0.599116	0.655354	0.723244
LHD	10000	0.107775	*32%*	*10%*	*44%*	*15%*	0.575457	0.814691	0.611363	0.674489	0.752153
LHHD	10000	0.229032	*8%*	*26%*	*27%*	*38%*	0.762835	0.511734	0.694663	0.612550	0.547798

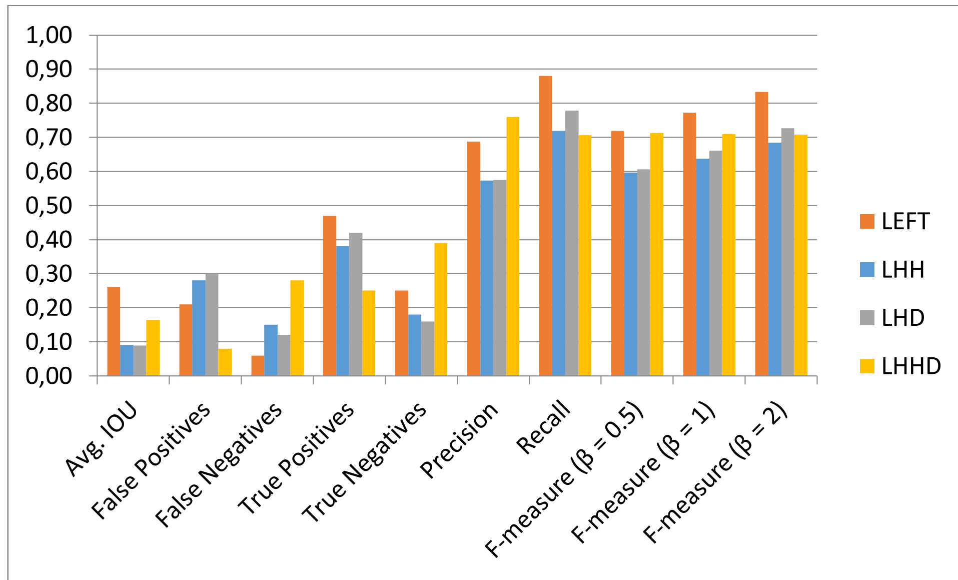


Table 5.2 – (Top) Statistics for the different trainings of YOLO after 10000 iterations. (Bottom) The results are also illustrated in a graph

5.2.2. Tracking with Head Detection

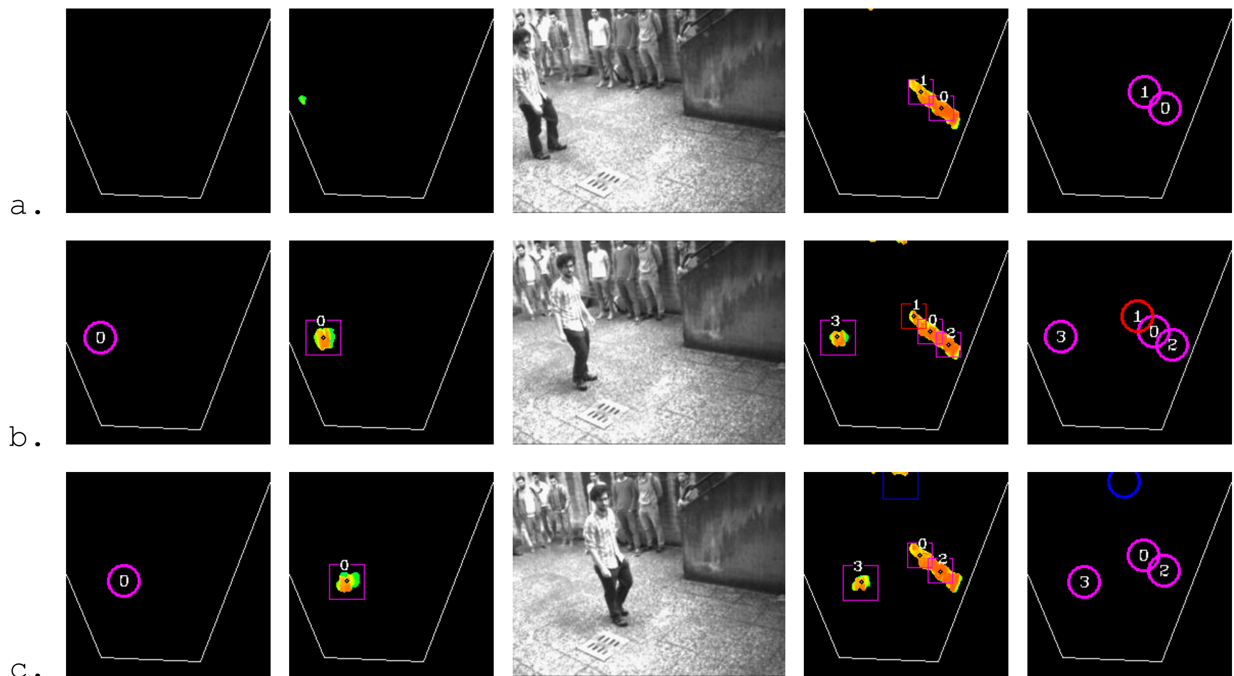
Using *YOLO* as described in *section 4.4* allows for a solution to a fundamental problem that affected *PeopleTracking*, that is confusing objects with people. In *figure 5.3*, a sequence is shown where a staircase with a high railing is present in the background. This peculiar object is not correctly filtered by the background subtraction and is detected by the original system as a row of people, because its dimensions match with the internal hand-crafted parameters that are used to analyse the top-view maps.

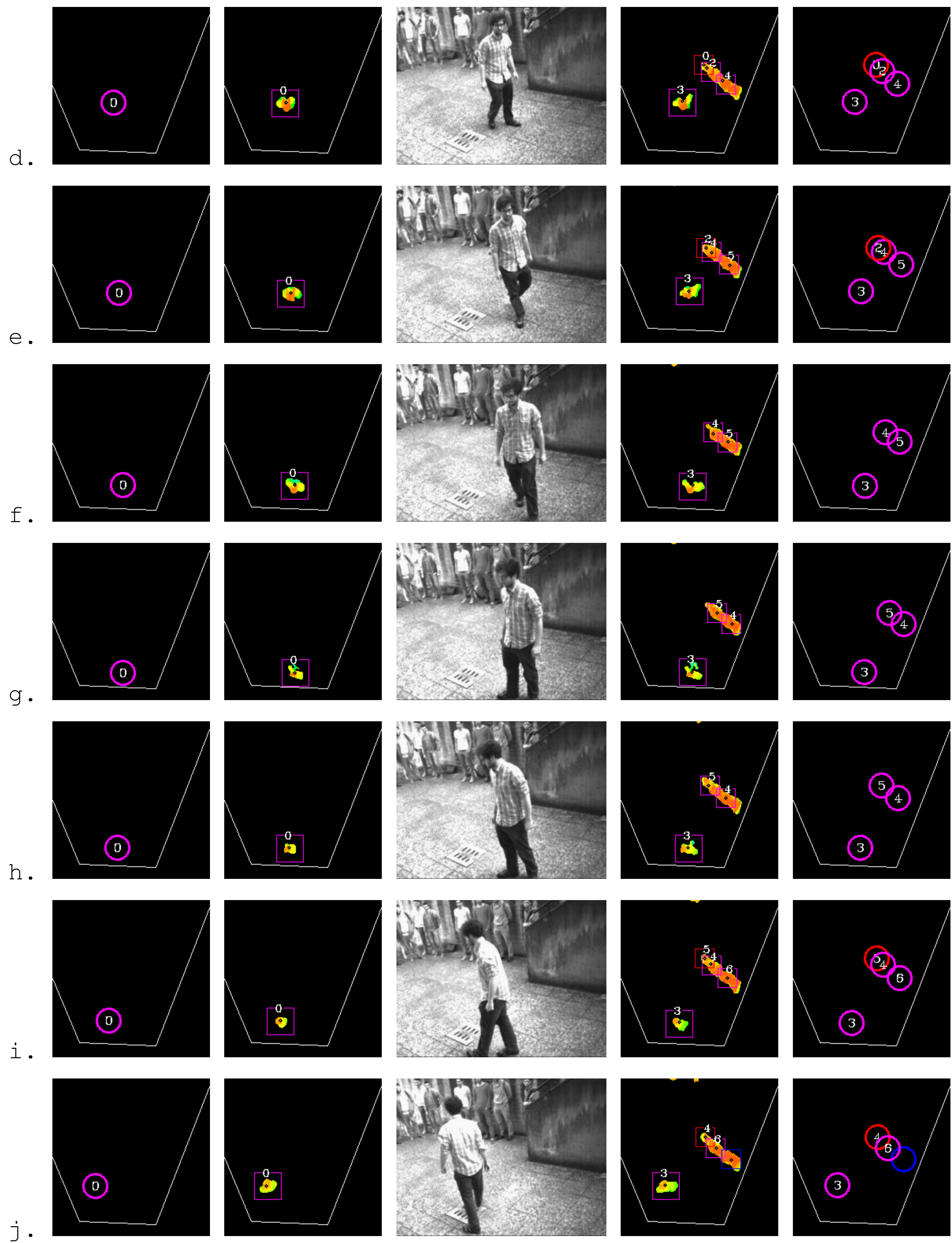
In the sequence, the resulting row of tracked subjects is not displayed as static, but they appear to be moving and occluding each

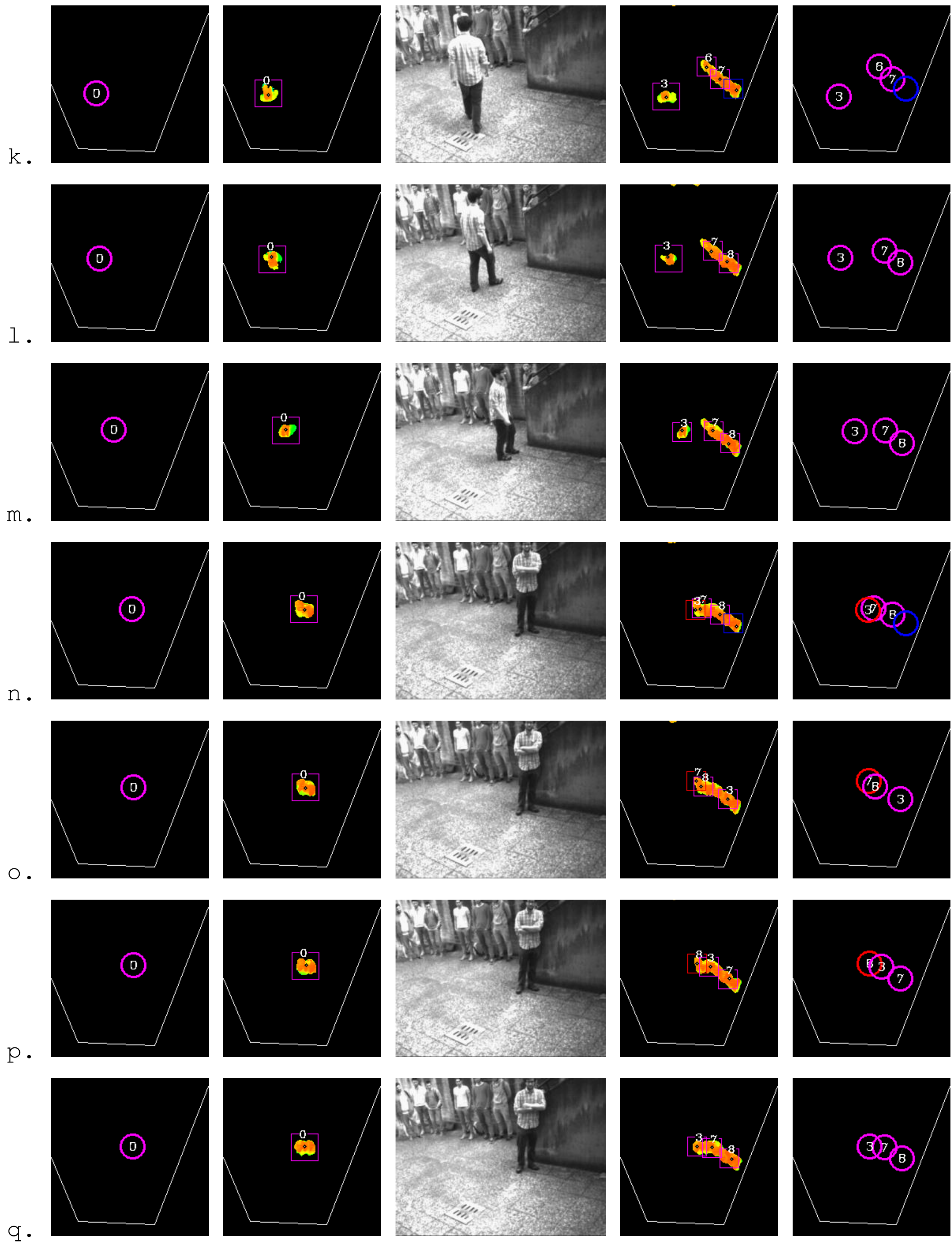
other, as can be seen by considering that the numbers identifying them are constantly changing in the frames shown below.

The problem gets worse when, in frame *n*, the actual person in the scene leans on the staircase, thus joining the row. In that situation, the person is extremely likely to match with one of the subjects that constitute the row during the second phase of the tracking algorithm (*measurement*). The effect of this circumstance is that the system loses track of the person's identity: when he enters the scene in frame *b*, he is assigned the descriptor number 3, upon getting close to the wall in frame *n* his descriptor is swapped with number 7, that is retained as he walks away from the staircase in frame *s*, thus compromising any attempt to keep track of its movements.

On the contrary, by filtering the top-view maps with *YOLO*'s predictions, the area that corresponds to the staircase is erased prior to the tracking algorithm's action. As a result, the enhanced system can accurately keep track of the person's position even in frame *n*.




d.
e.
f.
g.
h.
i.
j.


k.
l.
m.
n.
o.
p.
q.

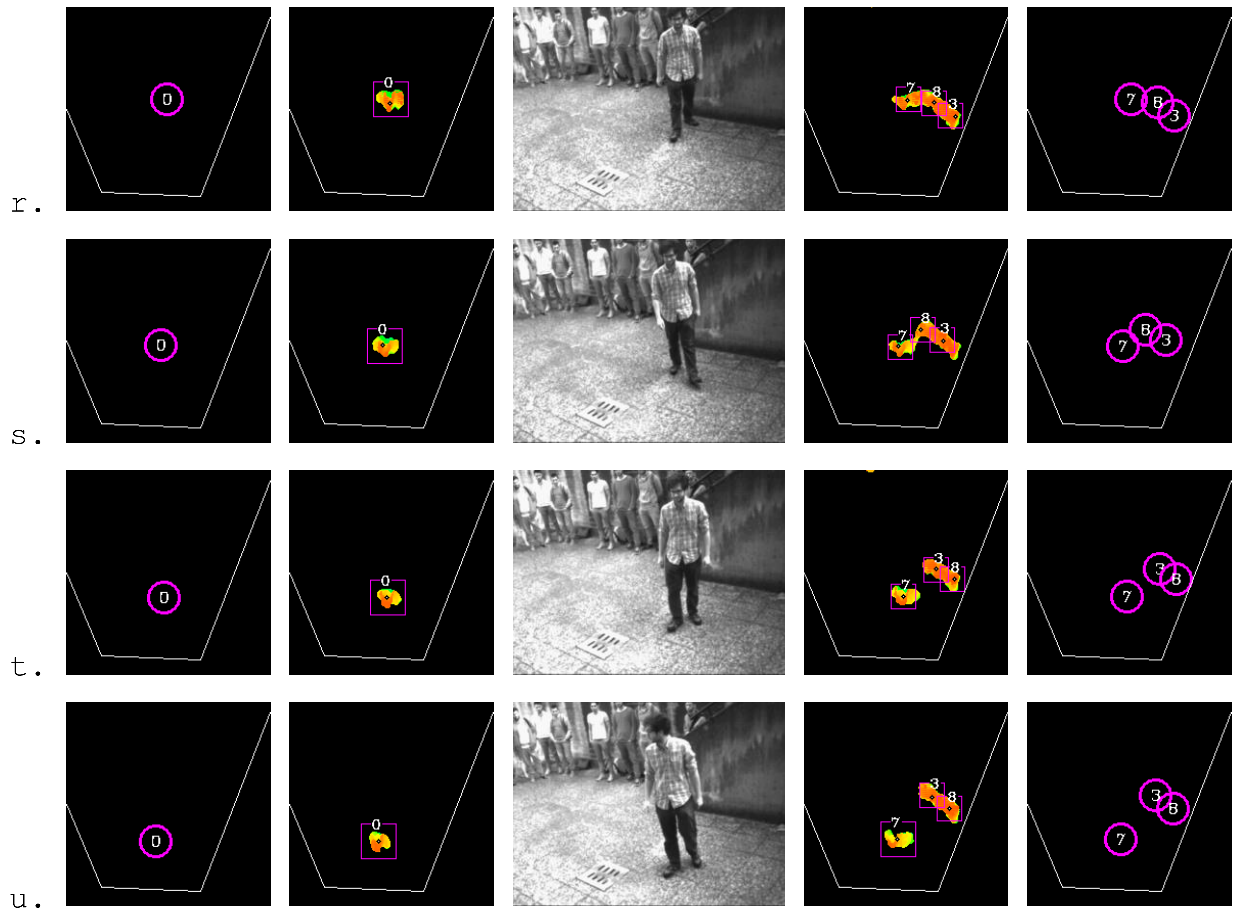


Figure 5.3 – Output of PeopleTracking with (left) and without (right) using YOLO for top-view maps filtering.

The test shown above uses the left-trained variant of *YOLO*, which proved to be more effective than the alternatives (see *section 5.2.1*). This also prevents the system's overall speed from further dropping, since the images acquired by the camera can be used by the network without any additional elaboration.

The introduction of *YOLO* in the pipeline, however, comes at a significant cost in terms of computational time. Due to the removal of the background subtraction procedure, which substantially reduced the points that had to be rototranslated to obtain the top-view maps, the time needed by our testing computer to calculate a single frame doubled, going from 36.18 ms (without *YOLO*) to 72.33 ms (with *YOLO*) on average.

Due to technological constraints, the tests were made using pre-calculated predictions from the neural network. In a real-world scenario where *YOLO* is supposed to predict positions while *PeopleTracking* is active, the overall time needed for the system to process a frame would be given by $T_f = T_y + T_{pt}$, where T_{pt} is the time that the tracking system needs to process a single frame (72.33 ms according to our measurements above) and T_y is the time that *YOLO* needs to make predictions for a frame. T_y is highly dependant on the hardware of the system we are using. In our tests, it varied from roughly 10 s when running on CPU to approximately 0.1 s when working with an NVIDIA Tesla C2070 graphical processing unit. According to *J. Redmon* in [15], *YOLO* can run faster on more modern GPUs.

6. Conclusions and Future Developments

In this thesis, the functionalities of an existing stereoscopic vision-based tracking system were extended.

The need of an external calibration was removed, thus drastically simplifying the system's set-up and making it independent of the surrounding environment. This is fundamental for any real-world application of our system where the camera might change its location. A deep-learning based software module was introduced to increase the precision of the tracking procedure. The technology at its core is extremely popular with the computer vision researchers and proves to be very effective in our system too.

During the testing of our system, the consideration was made that the head detection module might be used in the tracking algorithm by taking advantage of its predictions in place of the measured positions of the subject that are obtained by analysing the top-view maps. This modification would expectedly reduce the computational time for the algorithm, as its second and third phases (measurement and localisation) would become much easier, thus further improving the system.

7. Appendix: Configuring YOLO

7.1. Overview

This appendix features a brief description of the procedure that was followed to configure, train and test neural networks on the *Darknet* framework. The network used in this thesis is a slightly modified version of *YOLO*, which is described in *section 4.2*. Its configuration procedure is based on the instructions that can be found on Joseph Redmon’s website [16].

7.2. Downloading Darknet

Firstly, *Darknet* has to be downloaded from [16]. The folder already contains the necessary configuration files that define *YOLO* along with other networks. In order to have *YOLO* only detect a single class of objects, the configuration file */cfg/yolo.cfg* and the source files */src/yolo.c /src/yolo_demo.c /src/yolo_kernels.cu* are conveniently modified. The modified network was given the name *Hefi* (standing for Head Finder) and the files listed above were renamed accordingly. The new network was then properly interfaced with *Darknet* by editing /src/darknet.c.

7.3. Acquiring and Labelling images

The dataset is built using the RGB-D camera developed within the DISI with *SmartCamera*, a program also provided by the department that can be used to calibrate the sensor and acquire pictures. As a result, single-channel left images and three-channel images containing both the left and the disparity information are obtained.

The images have then to be labelled so as to obtain a precise description of what *Hefi* is expected to predict. During its training, these data are used by *Darknet* as a reference for the trial-and-error system that regulates its learning. Upon testing, they can be used to analyse *Hefi*’s output and determine its effectiveness. The

labelling process uses *HeadLabeller*, a simple program which lets the user draw a rectangle on the heads that are present in each image. The results of this process are saved in a csv file, which contains a line for each drawn box, including its position, its dimensions in pixels and the number of its image.
It was noticed that sometimes the operation of recording labels which are too close to the border of the picture may overflow, thus determining an extremely high number in the resulting file. In order to fix this problem, the output should be analysed and any number that exceeds the image's resolution should be subtracted from 65536 in order to obtain the correct value.

Figure 7.1 - An image labelled using HeadLabeller.

7.4. Preparing the Training Set

7.4.1. Preparing Files

While training, *Hefi* and *YOLO* require that the labels be contained in separate text files, one for each image of the training set, and that the position and dimension of boxes be relative to the dimension of the image and expressed with a floating point number. These

conversions are carried out by a utility program called *HefiConverter*.

In order to prepare the training set, the folder containing the training images and the one containing the labels should be placed in the same folder and their name should be identical, with the first one including the string “images” and the second one including the string “labels” in its place. For instance, the folders used for the trainings in this thesis were all contained in ~/data and their names were “images”, “labels”, “images_lhh”, “labels_lhh”, “images_lhd”, “labels_lhd”, “images_no_plane” and “labels_no_plane”. Furthermore, the images and the corresponding labels should have the same name (including the file extension). These naming conventions can easily be overridden by editing the source code of *Darknet*.
Since the sample datasets found on [14] do not include images that do not contain any prediction boxes, *HefiConverter* does not generate any file for these images. This means that images that are not supposed to contain any head and consequently do not match with a generated label file have to be removed.
Finally, a text file must be generated containing one line for each image that appears in the set, with its full path. An easy way to obtain it is using the Linux command *readlink –f* on the files in the images folder and redirecting its output to a file.
The location of the generated file, along with a folder that will contain the intermediate products of the training process have to be specified in <darknet-folder>/src/yolo.c (resp. <darknet-folder>/src/hefi.c). Currently, the training file is /home/<user>/train.txt and the folder is /home/<user>/backup.

7.4.2. Image Encodings

The images contained in the folder mentioned above have to be obtained from the data acquired as described in *section 7.3*. *HefiConverter* also includes the following set of image conversion routines:

- From single channel to three-channel white and black images
- From three-channel left and disparity to three-channel white and black images (with or without removing the walking plane)
- From three-channel left and disparity to three-channel LHH.
- From three-channel left and disparity to three-channel LHD.
- From three-channel left and disparity to three-channel HHD.

It must be noticed that, even when the training simply uses grayscale images, a conversion is needed to match the number of channels. The network can be altered to only take single channel images as input by properly editing <darknet-folder>/cfg/yolo.cfg (resp. <darknet-folder>/cfg/hefi.cfg).

7.5. Launching the training

After setting up the training set as described in the previous section, training can simply be initialised by compiling *Darknet* and using the following syntax: *darknet yolo train cfg/yolo.cfg <weights-file>* (resp. *darknet hefi train cfg/hefi.cfg <weights-file>*). The status of the network's training is stored in weights files. When the first training is launched, a proper file must be used which contains convolutional weights pre-trained on Imagenet and can be downloaded from [16]. If the training is interrupted, it can be resumed by using intermediate weights instead. These weights are stored in the folder that was specifically prepared in *section 7.4.1* and are be saved by *Darknet* after a fixed amount of training iterations. A complete training cycle requires 40000 iterations.

When a training process begins, data are shown on screen representing the network's guesses and its progresses. Make sure that the

displayed numbers generally correspond to valid floating point numbers between 0 and 1. If all numbers are *-nan*, make sure the instructions in *section 7.5* were followed thoroughly.

7.6. Testing

7.6.1. Preparing a Test Set

The network's training can be interrupted at any time to start a test, which will use the trained system to detect objects in the images of a test set. These images have to be obtained by using *HefiConverter* as described in *section 7.4.2*, then a text file must be generated containing one line for each image that is in the set, with its full path (*readlink* can again be used as described in *section 7.4.1*).

A training can be initialised by using *darknet yolo test cfg/yolo.cfg <weights-file> < <test-set-text-file>* (resp. *darknet hefi test cfg/hefi.cfg <weights-file> < <test-set-text-file>*). If *Darknet* was compiled using OPENCV, the test will show the results in a window, else it will save them in png files.

7.6.2. Alternative Testing Modes

Hefi includes some additional testing modes that are not originally available in *YOLO* and were developed to be used in this thesis:

- **pipe** prints the predictions on a pipe (whose other end is supposed to be used by *PeopleTracking*). For every image, a line is printed for every predicted box containing the coordinates of its centre and then an additional line is printed that marks the end of predictions for the current picture.
- **txtout** saves the predictions in a text file using the same representation as **pipe** and can be used for testing purposes.
- **extout** saves the predictions in a text file using a representation that matches the one used by *HeadLabeller*.

7.6.3. Comparing Results

The results of a test can be quantified by comparing the manually drawn labels from *HeadLabeller* with the corresponding predictions made by *Hefi* when operating in **extout** mode. In this thesis, we analysed the following statistics:

- **Intersection over Union (IOU)**, which is calculated for each frame by dividing the intersection of the boxes described by labels and predictions with their union.
- The number of **false positives (FP**), which is the count of the number of frames where heads are found by the network even though they had no corresponding labels.
- The number of **false negatives (FN)** or **missed frames**, which is the count of the number of frames where no head is found even though there are labels.

These statistics are computed by another utility program, called *LabelComparer*, which yields the IOU for each frame, the overall average IOU and the number of false positives and false negatives.

8. References

[1] A. Muscoloni, S. Mattoccia, "Real-time tracking with an embedded 3D camera with FPGA processing", International Conference on 3D Imaging (IC3D), Liège, December 2014.

[2] V. Poli, "Individuazione di superfici planari e sistemi di riferimento in nuvole di punti generate da un sistema 3D", bachelor thesis in Ingegneria Elettronica, AY 2013-2014.

[3] E. Golfieri, "Studio e valutazione di metodologie per la rilevazione di piani da nuvole di punti mediante la trasformata di Hough", bachelor thesis in Ingegneria Informatica, AY 2014-2015.

[4] M. Rucci, "Plane detection from pointclouds by means of a region growing approach", bachelor thesis in Ingegneria dell'Automazione, AY 2014-2015.

[5] D. Barchi, "Algoritmo per la segmentazione di piani da nuvola di punti basato su normali", bachelor thesis in Ingegneria dell'Automazione, AY 2014-2015.

[6] A. Garbugli, "Sperimentazione di algoritmi per l'analisi di nuvole di punti per applicazioni di guida autonoma", bachelor thesis in Ingegneria Informatica, AY 2015-2016.

[7] T. van Oosterhout, S. Bakkes, B. Kröse, "Head Detection in Stereo Data for People Counting and Segmentation", International Conference on Computer Vision Theory and Applications (VISAPP), Vilamoura, March 2011.

[8] P. Viola, M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features", IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), 2001.

[9] R. Girshick, J. Donahue, T. Darrell, J. Malik, "Region-based Convolutional Networks for Accurate Object Detection and Segmentation", IEEE Transaction on Pattern Analysis and Machine Intelligence (PAMI), 2012.

[10] T. Vu, A. Osokin, I. Laptev, "Context-Aware CNNs for person head detection", International Conference on Computer Vision (ICCV), Santiago, December 2015.

[11] J. Redmon, S. Divvala, R. Girshick, A. Faradi, "You Only Look Once: Unified, Real-Time Object Detection", IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[12] J. Redmon, "Darknet: Open Source Neural Networks in C", http://pjreddie.com/darknet/, 2013-2016.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", CoRR, abs/1409.4842, 2014.

[14] S. Gupta, R. Girshick, P. Arbeláez, J. Malik, "Learning Rich Features from RGB-D Images for Object Detection and Segmentation", European Conference on Computer Vision (ECCV), Zürich, September 2014.

[15] J. Redmon, "Hardware Guide: Neural Networks on GPUs", http://pjreddie.com/darknet/hardware-guide/.

[16] J. Redmon, "YOLO: Real-Time Object Detection", http://pjreddie.com/darknet/yolo/.