UNIVERSITAT DE BARCELONA

COMPUTER SCIENCE BACHELOR'S THESIS

Faculty of Mathematics and Computer Science

A follow-me algorithm for AR.Drone using MobileNet-SSD and PID control

---

Author: Júlia Garriga Ferrer

Director: Dr. Lluís Garrido

Affiliation: Departament of Applied Mathematics and Analysis

Barcelona, June 27, 2018

Abstract — In recent years the industry of quadcopters has experimented a boost. The appearance of inexpensive drones has led to the growth of the recreational use of this vehicles, which opens the door to the creation of new applications and technologies. This thesis presents a vision-based autonomous control system for an AR.Drone 2.0. A tracking algorithm is developed using onboard vision systems without relying on additional external inputs. In particular, the tracking algorithm is the combination of a trained MobileNet-SSD object detector and a KCF tracker. The noise induced by the tracker is decreased with a Kalman filter. Furthermore, PID controllers are implemented for the motion control of the quadcopter, which process the output of the tracking algorithm to move the drone to the desired position. The final implementation was tested indoors and the system yields acceptable results.

Keywords: Quadcopter, Object Tracking, PID control, MobileNet-SSD

Contents

Introduction

Motivation

Every year the industry of automation and robots is increasing and the business of UAV (Unmanned aerial vehicles) provides less expensive and better models. In particular, quadcopters or, as popularly said, drones, are becoming more readily available, smaller and lighter. At the same time, the industry of cameras is also inflating, thanks to applications like *Instagram* people create images and videos for the sole purpose of sharing them to the social media. Because of this, quadcopters are being used for the creation of spectacular aerial images and videos that could not be made before. But drones with high quality cameras are much more expensive than regular drones. Although now, some of these cheap quadcopters allow a camera to be inserted on top, which makes the overall price less expensive. Another technology that has been booming for two years now is the follow-me drones: quadcopters programmed to automatically follow a target around, giving the opportunity to film unique aerial shots. This technology can be created with the use of a GPS device along with a transmitter, or by using sensors and object recognition on the target. If we combine the follow-me technology, plus the cheap quadcopter, plus an inserted good camera, we get a not very expensive and useful device, that can create high resolution videos while following you around.

The motivation of this project is to create a follow-me quadcopter implementation, able to track a target through daily activities, like running, climbing, swimming, etc. by only using the images obtained from the drone, with the help of computer vision algorithms.

Objectives

The objectives of this thesis are three. First, to implement a tracking algorithm in the three-dimensional space from two-dimensional video frames, this is, an algorithm able to follow the movement of the person through the height, width and depth planes using consecutive frames from a video. Second, to control an AR.Drone 2.0 so that it follows the observations given by the tracking algorithm, creating a follow-me implementation of the AR.Drone 2.0. Third, to run the implementation in a low speed processor: the CPU of a laptop, or even a smartphone or tablet, allowing the target to send, in real time, commands to the quadcopter.

Next a brief explanation of the main technologies will be introduced.

MobileNet and Single Shot Multibox Detector (SSD)

In general, the input of an object tracker is the bounding box containing the object to track. To obtain this bounding box an already trained object detector network can be used. In this case we use an implementation of the MobileNet-SSD detection network.

On one hand, MobileNet, [HZC+17], is an efficient network architecture especially designed for mobile and embedded vision applications. MobileNets are small, low-power, low-latency models effective across a wide range of applications and use cases including object detection, classification, face attributes and large scale geo-localization. The accuracy of MobileNets is surprisingly high and good enough for many applications, although not as good as a full-fledged neural network.

On the other hand, SSD, [LAE+15], is a method for detecting objects using a single deep neural network, easy to train and simple to integrate in systems that require a detection component. Moreover, experimental results on the PASCAL VOC, COCO, and ILSVRC datasets show that SSD has competitive accuracy to other slower methods. This combination of speed and accuracy make this method a very good option to use for our purpose.

PID control

To accomplish the task of automatic control of an AR.Drone 2.0 a PID controller is required.

The proportional-integral-derivative controller, or PID controller, is the most common type of controller used for UAV stabilization and autonomous control. It is a control loop feedback mechanism that attempts to minimize the error between a measured value and a desired value. The three terms: proportional, integral and derivative, compose the controller algorithm and try to minimize this error. The proportional corrects instances of error, the integral corrects accumulation of error, and the derivative corrects the actual error versus the error from the last iteration. To obtain a stable PID controller three parameters related to each of these terms have to be tuned. The goal of tuning is to reach the point right before erratic behaviour, where the quadcopter can get to the desired state quickly but without overshooting or oscillations. The parameters that produce the desired behaviour depend on the dynamics of the system being controlled.

Thesis organization

This essay is organized in five chapters:

- Chapter 1 explains the preliminaries needed to start with this project. First, it explains the different state of the art object detectors and object trackers, starting with an introduction to neural networks. Second, it talks about the quadcopter chosen for the project: the AR Drone 2.0, and the different methods that exist to create a communication with this quadcopter.
- Chapter 2 gives an in depth explanation of the project's framework, mostly theoretical without entering in the coded software. It introduces the Kalman filter for smoothing the tracking output and the PID controller for the control of the quadcopter.
- Chapter 3 describes, given the theoretical framework from Chapter 2, the implementation of the tracking and moving algorithms.
- Chapter 4 shows the results obtained of the implementation.
- Chapter 5 concludes the thesis and talks about future work.

1 Prior knowledge

The desired quadcopter behaviour is achieved by the interaction of two separate modules which provide two different functionalities: the tracking of the person on one hand, and the actual control logic of the quadcopter on the other. For the first task an object detection system is needed in order to find the person in the first frame right after the drone startup, and a tracking system is needed in order to follow this detected target. For the second task a control API can be made from scratch using the quadcopter low-level commands, or an existing, off-the-shelf and ready-to-use library can be used. The goal of this chapter is to provide a succinct introduction to the topics that have to be addressed in order to implement a solution for both tasks, and at the same time, to review the most relevant approaches to date and decide which are the most convenient for this thesis.

1.1 Computer vision preliminaries

The problem of human detection (embedded in the general problem of object detection) is the problem of automatically locating people in an image or video sequence, the latter case is usually referred to object tracking. When dealing with a video feed, there are two approaches to locate the interest object in each frame. On the one hand an object detector can be queried on every frame, so that the video sequence object tracking is effectively reduced to multiple independent (per-frame) detections. On the other hand a time-aware system can be used so that prior information from past frames is exploited to infer the object location in the current frame. A memoryless approach such as the former is usually more computationally expensive because the object location must be determined every time from scratch, while the latter approach tends to be faster but less accurate because the prior information alone (such as the previous object location) has to be corrected by some ad hoc hypothesis or model, which tends to produce drifted predicted locations. In this section we will focus on the general problem of object detection and tracking, starting with a brief explanation of how do supervised learning and neural networks work, which will help us understand the now state of the art object detection systems.

1.1.1 Supervised learning and neural networks

Supervised learning is a statistical subject which provides the mathematical setting to learn from example. Specifically, one has a set of *training* samples

$$D = \{(\mathbf{x_i}, \mathbf{y_i}) \mid \mathbf{x_i} \in \mathbb{R}^n,\ \mathbf{y_i} \in \mathbb{R}^m,\ i = 1, ..., N\}$$

drawn from the joint unknown distribution

$$f : \mathbb{R}^n \times \mathbb{R}^m \longrightarrow [0,1]$$

which models a stochastic function

$$\begin{aligned} h &: \mathbb{R}^n \longrightarrow \mathbb{R}^m \\ \mathbf{x} &\longmapsto \mathbf{y} = h(\mathbf{x}) \end{aligned} \tag{1.1}$$

Given a parametric model

$$\begin{aligned} \hat{h} &: \mathbb{R}^n \times \mathbb{R}^k \longrightarrow \mathbb{R}^m \\ (\mathbf{x}, \psi) &\longmapsto \hat{h}(\mathbf{x}, \psi) \end{aligned}$$

and a *cost* function

$$\begin{aligned} L &: \mathbb{R}^m \times \mathbb{R}^m \longrightarrow \mathbb{R} \\ (\mathbf{a}, \mathbf{b}) &\longmapsto L(\mathbf{a}, \mathbf{b}) \end{aligned}$$

the goal of supervised learning is to find

$$\hat{\psi} = \arg\min_{\psi} \sum_{i=1}^{N} L(\hat{h}(\mathbf{x_i}, \psi), \mathbf{y_i}) \tag{1.2}$$

so that $\hat{h}(\cdot, \hat{\psi})$ is a good approximation of h. Supervised learning can be divided into two categories:

- Classification problems: discrete output, it includes models such as Support Vector Machines, Artificial Neural Networks and Naïve Bayes classifiers.
- Regression problems: continuous output, it includes models such as Linear Regressors, Decision Trees and Artificial Neural Networks.

Since Artificial Neural Networks currently provide state-of-the-art results in supervised learning tasks we will stick to these models.

Artificial neural networks are a brain-inspired system intended to replicate the way the human brain works. They consists in a collection of interconnected units or nodes called artificial neurons that can transmit signals from one to another and operate in parallel according to the given input. Depending on their inputs and outputs, these neurons are generally arranged into three different layers (fig 1.1):

- Input layer: dimensioned according to the input.
- Hidden layer(s).
- Output layer: dimensioned to fit the proper output.

Depending on the connectivity between the neurons in the hidden layer(s) the neural network can be a feed-forward network, where the information travels in one direction, from input to output, or a feedback network, where the information can travel in both directions.

Neural networks can also be used for unsupervised learning (e.g. autoencoders, [Bal12]), but our focus will be on the supervised neural networks, which are the most common. From now on, when referring to neural networks we will be talking of supervised neural networks. As neural networks are a type of supervised learning method they will have a training dataset, a parametric function (which is the neural network) and a cost function, as explained above, and their goal will be to find the ψ that meets (1.2). This ψ is a k-dimensional vector and their $\psi_1, .., \psi_k$ components are called weights, ω, and biases, b. A basic neural network with no hidden layers is

$$\hat{h}_i = \sum_{j=1}^{n} \omega_{ji} x_j + b_i \qquad i = 1, ..., m;\ b_i, \omega_{ij}, x_j \in \mathbb{R}$$

where ω_{ij}'s are the weights and b_i's are the biases, so that $\omega_{ij}, b_i \in \{\psi_1, .., \psi_k\}$.

In figure 1.1 we have an example of a neural network with two hidden layers of dimensions 3 and 4, with all the corresponding weights and biases.

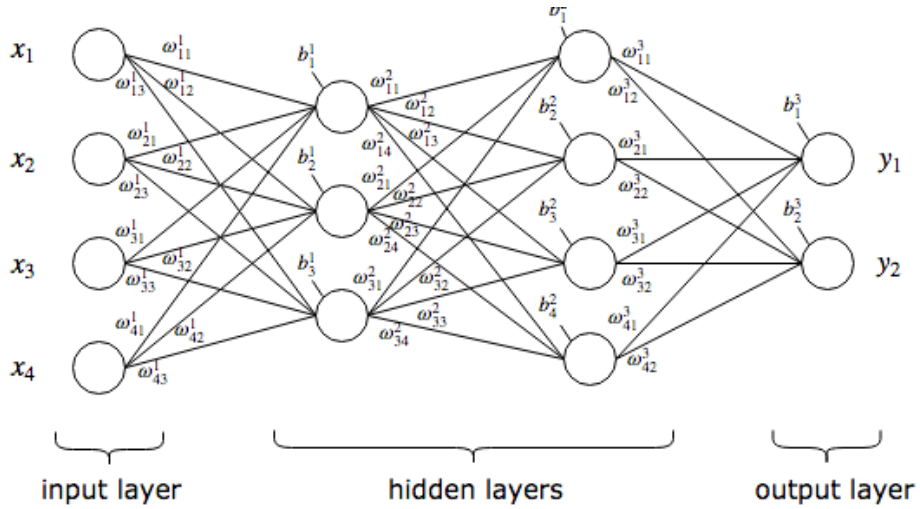


Figure 1.1: Fully connected 4-layer neural network.

Let M_l be the matrix formed by all the weights from layer $l-1$ of dimension v to layer l of dimension u:

$$M_l = \begin{pmatrix} \omega^l_{11} & \omega^l_{21} & \cdots & \omega^l_{(v-1)1} & \omega^l_{v1} \\ \omega^l_{12} & \omega^l_{22} & \cdots & \omega^l_{(v-1)2} & \omega^l_{v2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \omega^l_{1(u-1)} & \omega^l_{2(u-1)} & \cdots & \omega^l_{(v-1)(u-1)} & \omega^l_{v(u-1)} \\ \omega^l_{1u} & \omega^l_{2u} & \cdots & \omega^l_{(v-1)u} & \omega^l_{vu} \end{pmatrix},$$

So that with no hidden layers we have:

$$\hat{h} = Mx + b, \tag{1.3}$$

with $x \in \mathbb{R}^n, b \in \mathbb{R}^m, M = M_1 \in \mathbb{R}^{m \times n}$.
If we had one hidden layer of dimension d our output would be:

$$\hat{h} = M_2(M_1 x^1 + b^1) + b^2 = (M_2 M_1)x + (M_2 b^1 + b^2) = Mx + b, \tag{1.4}$$

with $b^1 \in \mathbb{R}^d,\ b^2 \in \mathbb{R}^m,\ M_1 \in \mathbb{R}^{d \times n},\ M_2 \in \mathbb{R}^{m \times d}$.
Equation (1.4) shows that, with this network architecture, and even by adding additional hidden layers, the only functions that our network will learn are going to be linear functions. To be able to approximate nonlinear functions we need (non-linear) activation functions. For example, if our network is used to know if tomorrow is going to rain (output 1 if so, 0 if not) we could write something like this:

$$\hat{h}_i = \begin{cases} 0 & if \ \sum_{j=1}^n \omega_{ij} x_j \leq \xi_i \\ 1 & if \ \sum_{j=1}^n \omega_{ij} x_j > \xi_i \end{cases} \qquad i = 1, ..., m \qquad \xi_i, \omega_{ij}, x_j \in \mathbb{R}$$

In this case, the activation function used is called the threshold function, which generates the output 1 if the input exceeds a certain value $\xi_i = -b_i$. This type of neural networks that performs binary classification are called perceptrons.
But perceptrons are not continous (a small change in the input may produce a large change in the output), therefore expression (1.2) cannot be solved using standard analysis techniques. To fix this, other activation functions, which are differentiable, are used. For example the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$

which applied to our neural network would be:

$$\hat{h}_i = \frac{1}{1 + exp(-\sum_{j=1}^n \omega_{ij} x_j - b_i)} \qquad i = 1, ..., m \qquad b_i, \omega_{ij}, x_j \in \mathbb{R},$$

the hyperbolic tangent function:

$$f(x) = tanh(x) \in (-1, 1)$$

so that

$$\hat{h}_i = tanh\left(\sum_{j=1}^{n} \omega_{ij} x_j + b_i\right) \qquad i = 1, ..., m \qquad b_i, \omega_{ij}, x_j \in \mathbb{R}$$

or the Rectified Linear Unit (ReLu) function:

$$f(x) = max(0, x)$$

so that

$$\hat{h}_i = max(0, \sum_{j=1}^{n} \omega_{ij} x_j + b_i) \qquad i = 1, ..., m \qquad b_i, \omega_{ij}, x_j \in \mathbb{R}.$$

Now that we have the equations of the neural network, we need an algorithm that learns the weights and biases so that the output from the network approximates $\mathbf{y_i}$ for all training inputs $\mathbf{x_i}$. Here is where the cost function is introduced, a typical one is:

$$C(\psi) \equiv \frac{1}{2n} \sum_{x} \|\hat{h}(\mathbf{x_i}, \psi) - \mathbf{y_i}\|^2.$$

C is called the quadratic cost function and is the mean squared error between the real and the desired output. The aim of our training algorithm is to minimize this cost function and the algorithm used is gradient descent, which repeatedly computes the gradient $\nabla_\psi C$, normally by means of the backpropagation algorithm [EREHJW86], and moves proportionally to it's opposite orientation until a local minimum is reached.

Neural networks have many architectures and classes, such as the explained feed-forward and feedback. This project uses neural networks to find persons in an image, for which a certain class of neural networks is used: Convolutional Neural Networks (ConvNets or CNNs). CNNs are a category of deep feed-forward neural networks that have proven very effective in areas such as image recognition and classification.

Before explaining CNNs we will make a small introduction on Multilayer Perceptrons (MLPs), of which CNNs are inspired from. The multilayer perceptron is a specific feed-forward neural network architecture, with at least one fully connected layer (a part from the input and output layers) and one non-linear activation function. MLPs use backpropagation for training the network and can distinguish data that is not linearly separable. Their multiple layers and the activation function is what discern them from perceptrons.

Suppose now that our inputs are images, so that the pixels of the images compose the input layer, and that our goal is to learn features from a dataset of images to classify them. If our images are small, say 28×28 pixels of images like the MNIST dataset [LC10], it is computationally possible to learn weights on the entire image using fully connected layers. However, with larger images, say 96×96 images, learning weights with fully connected layers of the entire image is very computationally expensive, $\sim 10^4$ input pixels, and for 100 neurons in the single hidden layer the parameters

to learn would be $\sim 10^6$. The feedforward and backpropagation algorithm would also increase they learning time approximately $\sim 10^2$. To solve this, a plausible solution is restricting the connections between the hidden units and the input units, see 1.1, by removing the fully-connection and allowing each hidden unit to connect to only a small subset of the input units.

CNNs are neural networks inspired by MLPs which were developed originally to process images, and that can obtain useful information by how the pixels are located through the image. For example, if the input images are of 32×32 pixels, and they have 3 channels, the ConvNet input will be a $32 \times 32 \times 3$ array of pixels. As the name suggest, all CNNs are composed of (at least) one or more convolutional layers, which apply convolutions to the image [GBC16, LBBH98].

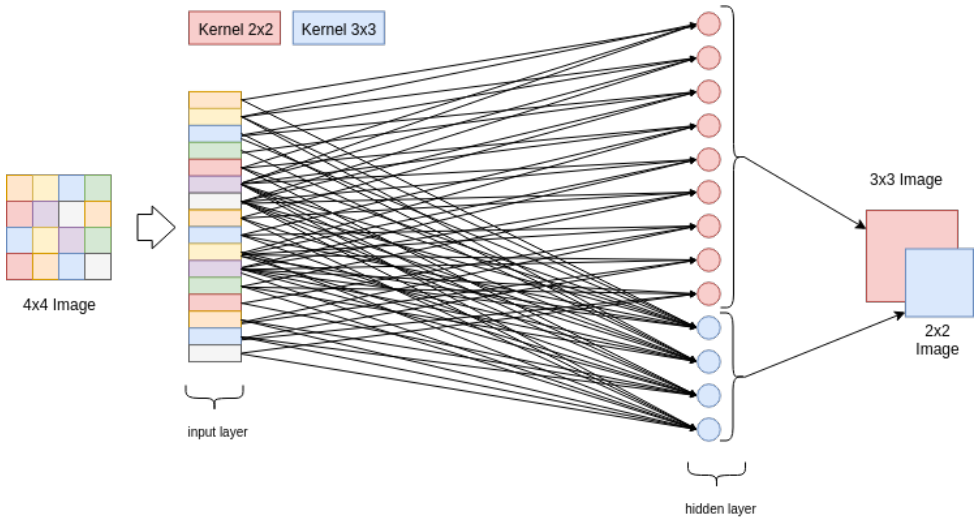


Figure 1.2: Convolutional layer on a 4×4 image with 1 channel, and with a 2×2 and a 3×3 kernel with stride 1. The links show how the pixels contribute to the final convolution.

Each convolutional layer applies different kernels to compute the convolutions, and this set of kernels compose our weights. In our last example, one kernel could be of dimensions $5 \times 5 \times 3$, so that its depth matches the inputs depth. In figure 1.2 we have an example of a convolutional layer, with two different kernels applied, although normally all kernels of a layer have the same dimensions.
In a traditional CNN architecture there are other layers besides the convolutional layers such as fully connected layers or pooling layers.
Once obtained the weights using convolution, we could use them to classify the images, for example with a softmax classifier [KSH12], but this can be computationally challenging due to the big number of weights. A pooling layer is used to reduce dimensionality.
The two most common pooling functions are the average pooling, that computes the mean value of a region of the convolved image, and the max pooling, that computes the max value of a region of the convolved image. Figure 1.3 shows an example of a max pooling layer.

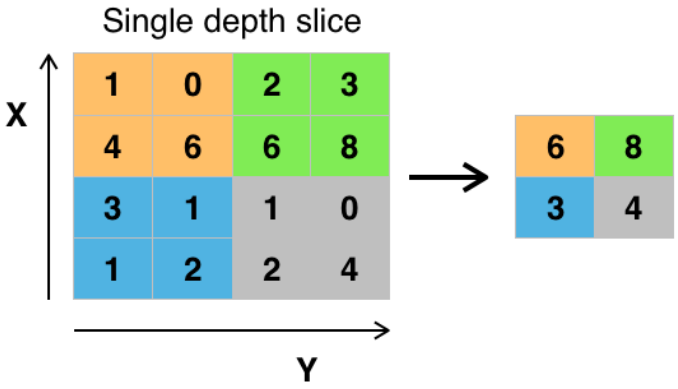


Figure 1.3: Max pooling with a 2×2 filter and stride 2. Image retrieved from [Com16a]

Summarizing, a convolutional neural network is comprised of one or more convolutional layers with nonlinear activation functions, alternated with some pooling layers, and then followed by one or more fully connected layers as in a MLP. They are designed to take advantage of the 2D structure of an image, easy to train and have many fewer parameters than the fully connected with the same number of hidden layers, see 1.4.

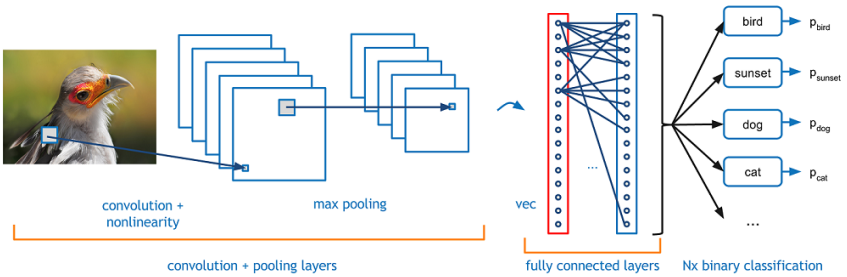


Figure 1.4: Example of a CNN architecture retrieved from [Des16].

1.1.2 Object detection

Every object has its own features that make it special or different from others (for example, all circles are round). These features are used for classifying an object into one or many different categories. But what if the object is not the whole image, but in a part of it? Here is where comes localization, which finds exactly where the object is, drawing the bounding box that contains it. Finally, we find that in the image there may be not only one object but many, and we want to classify and locate all of them. The problem of finding and classifying a variable number of objects in an image is what we will call object detection[1]. While the field of classification is practically solved, object detection has problems and challenges that have been tackled for the past years with the use of deep learning, and different approaches have been implemented to find the balance between accuracy and speed.

[1]Some communities use the term object recognition as the problem of classifying the detected objects and define object detection as only the technique of localizing the object in the image

We can say that the problem of locating and classifying multiple objects in an image, called object detection, is nowadays best solved with neural networks. But before the arrival of deep learning there existed other methods to tackle this problem, the two most popular ones were the Viola-Jones framework [VJ01] and the Histogram Of Gradients (HOG) [DT05]. The first method is fast and simple, implemented in point-and-shoot cameras. It works by generating different simple binary classifiers using Haar features. Even though it offers real-time performance and scale/location invariance, it has a few disadvantages like intolerance to rotation, sensitivity to illumination variations, etc. The second method counts occurrences of gradient orientation in localized portions of the image (cells) and groups them into a number of orientation bins, so stronger gradients contribute more weights to their bins and effects of small random orientations due to noice are reduced. HOG is used for extracting the features of the images, and is normally paired with a Linear Support Vector Machine[2] to classify them. Even though this method is superior than Viola-Jones, it is also much slower.

A few years ago, with the introduction of CNNs, researchers started developing methods for object detection using deep learning which lead to much more successful results than when using classical methods. Deep learning is a class of machine learning that uses a cascade of multiple layers with nonlinear functions for feature extraction and transformation. After having introduced neural networks and, particulary, CNNs in section 1.1.1, we will do an overview of the different state of the art approaches for object detection with the use of deep learning, and decide our best fit for this project. One of the first good methods developed for object detection using deep learning was Overfeat [SEZ+13], published in 2013, where they proposed a multi-scale sliding window algorithm using CNNs. Shortly after Overfeat, Regions of CNN features or R-CNN were introduced [GDDM13], which used a region proposal method (like selective search [UvdSGS13]) for extracting possible objects, then extracted features from each region with CNNs, and finally made the classification with SVMs. Although with great results, the training had a lot of problems. That is why a year later the same author published Fast R-CNN [Gir15]. Instead of applying CNNs independently in each region proposed by selective search and classifying with SVMs, the latter applied the CNN on the complete image and then used a region of interest (RoI) pooling layer [Gir15] with a final feed-forward network for classification and regression. This approach was faster, and with the RoI pooling layer and the fully connected layers the model became end-to-end differentiable and easier to train, but it still relied on selective search which was a problem when using it for inference. Shortly, *You Only Look Once: Unified, Real-Time Object Detection* (YOLO) paper by Joseph Redmon [RDGF15] was introduced, which proposed a simple CNN approach with great results and high speed allowing, for the first time, real time object detection. Instead of applying the model to an image at multiple locations and scales, they applied a single neural network to the full image, which divides the image into regions and predicts bounding boxes and probabilities for each region. Making predictions with a single network evaluation made it extremely fast, 1000x faster than R-CNN and 100x

[2]Support vector machines or SVMs [HDO+98] are supervised learning models for classifying that, given the labeled training data, the algorithm outputs a separating hyperplane that categorizes new examples

faster than Fast R-CNN. Later on, Faster R-CNN by Shaoqing Ren was published [RHGS15], which, following on the work of [GDDM13] and [Gir15], added a Region Proposal Network (RPN) to get rid of selective search and to make the model completely trainable, from end to end. RPN ranks region boxes (anchors) and proposes the ones most likely containing objects, so that the time cost of generating region proposals is much smaller with this method than with selective search. Finally, we have two notable methods: Single Shot Detector (SSD) which takes on YOLO by using multiple sized convolutional feature maps, achieving better results and speed [LAE+15], and Region-based Fully Convolutional Networks (R-FCN) [DLHS16] which follows on R-CNN methods but replacing the fully connected layers for fully convolutional ones and building strong region-based position-sensitive classifiers, which increase speed and achieve the same accuracy as Faster R-CNN.

For this project we need a detector that is able to detect a person rapidly in a simple CPU, and maybe in a smartphone or tablet. Each of these object detection techniques use a base network architecture at the early network layers, called feature extractor. These feature extractors change the final behaviour of the detector in terms of speed and accuracy.

Figure 1.5 shows the performance of some of the mentioned state of the art object detectors when using different feature extractors.

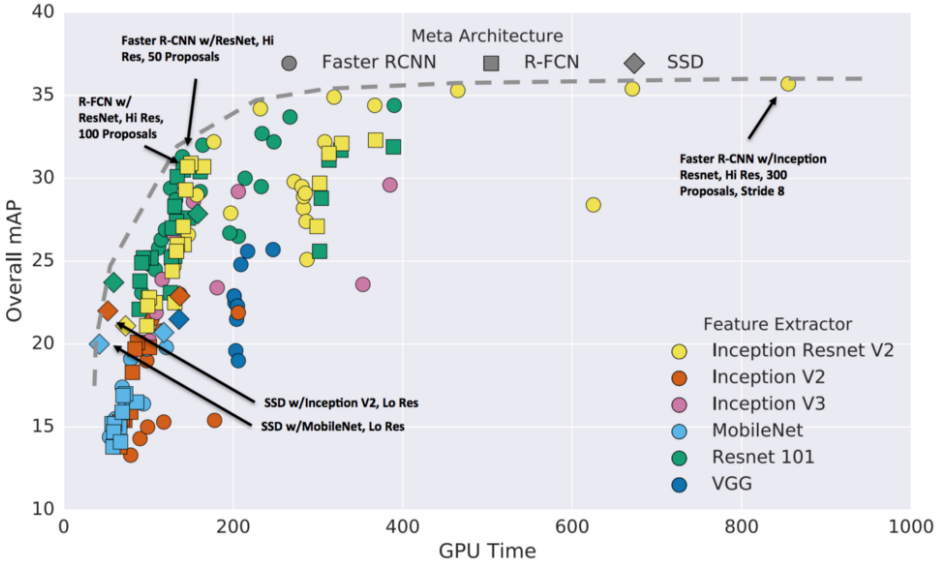


Figure 1.5: Object detectors comparision trained with MS COCO dataset retrieved from [HRS+16a]

While Faster R-CNN with Inception Resnet-based architecture [RHGS15, SIV16] is top 1 in accuracy, it implies a big loss of speed. For our purpose the best approach is using SSDs with MobileNet [HZC+17] or with Inception V2 [IS15], which still have a 20 mAP (mean average precision) of accuracy and their GPU time is a lot less than the others.

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Application was presented in

2017 as an innovative class of efficient models created to be used in mobile and embbeded vision applications. The MobileNet architecture is based on depthwise separable convolutions, which are a form of factorized convolutions, [WLF16], that factorize a standard convolution into a depthwise convolution and a 1×1 convolution (pointwise convolution). In a depthwise convolution the kernels of each filter are applied separately to each channel and the outputs are concatenated.

The MobileNet structure is built on depthwise separable convolutions, except for the first layer which is a full convolution. All layers are followed by a batchnorm [IS15] and a ReLu nonlinearity, except for the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification. Figure 1.6 compares a standard convolutional layer to the factorized layer with depthwise and pointwise convolutions.

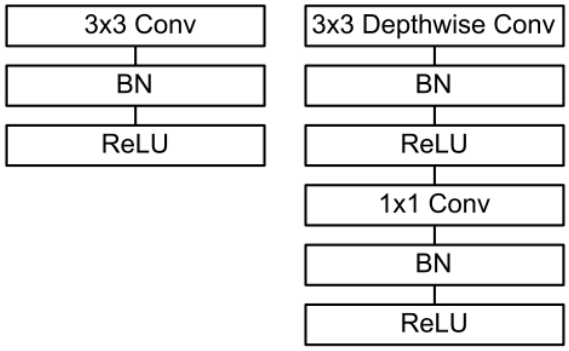


Figure 1.6: Left: a standard convolutional layer. Right: Depthwise Separable convolutions. Retrieved from [WLF16].

SSD: Single Shot Multibox Detector paper [LAE+15] was released at the end of November 2016 and reached new records in terms of performance and precision for object detection, scoring over 74 % mAP at 59 fps on standard datasets such as PascalVOC and COCO. The name Single Shot means that, like in YOLO, the tasks of object localization and classification are done in a single forward pass of the network, simultaneously predicting the bounding box and the class as it processes the image.

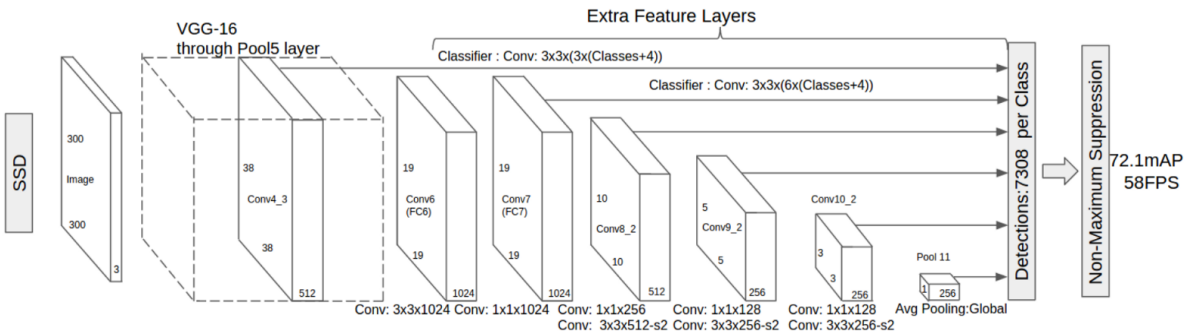
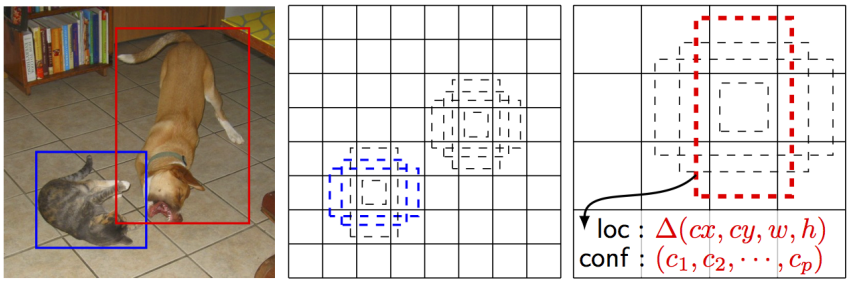


Figure 1.7: SSD architecture retrieved from [LAE+15]

The SSD approach, shown in figure 1.7, is based on a feed-forward convolutional network that produces a fixed number of bounding boxes and scores for the presence of object class instances in those boxes, finishing with a non-maximum suppression step to group together highly-overlapping boxes into a single box [HBS17, NG06]. The early network layers are based on a standard architecture used for high-quality image classification (truncated before any classification layers). In this project the base architecture is MobileNet, but in the original paper they use VGG-16. After the base architecture, a set of convolutional feature layers is added, which decrease progressively in size. Instead of only using each feature map as input for the next feature layer, they reshape this feature map into a vector, making the output be the join of all these vectors. This allows predictions of detections at multiple scales, for each feature map has information in a different, each time bigger, region of the image.



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

Figure 1.8: Matching of default boxes with ground truth boxes retrieved from [LAE+15]

During training, the ground truth information needs to be assigned to specific outputs in the fixed set of detector outputs. This means that some of the default bounding boxes have to be assigned to their corresponding ground truth detection, and the network has to be trained accordingly, see 1.8. Here is where they use Multibox, by matching each ground truth box to the default box (which vary over location, aspect ratio, and scale) with the best jaccard overlap [Jac01]. Unlike Multibox, they match default boxes to any ground truth with jaccard overlap higher than a threshold, simplifying the learning problem by allowing the network to predict higher scores for multiple overlapping boxes. The SSD training objective is derived from the Multibox objective but extended to handle multiple object categories. They use Smooth L1 location loss, which measures how the network predicted bounding boxes and the ground truth ones differ, between the predicted box and the ground truth box parameters, and a softmax loss for the confidence loss, which measures the reliance of the network to have an object inside a bounding box, so that the overall objective loss function is a weighted sum of the localization loss and the confidence loss:

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)),$$

where N is the number of matched default boxes, l the predicted box, g the ground truth parameters, and c the set of classes confidences.
When training, as most bounding boxes will have a low jaccard overlap, they will be interpreted as

negative training examples. This imbalance between the number of positive and negative examples is improved by using only a part of the negative examples (sorted using the highest confidence loss for each default box) so that the ratio between positive and negative examples is 3:1. Finally, to make the model more robust to various input object sizes and shapes, they randomly sample each training image, augmenting the training data. The non-maximum suppression applied at the end of the network is essential to prune the large number of boxes generated: filtering by the confidence loss and the jaccard overlap ensures that only the most likely predictions are retained by the network.

1.1.3 Object tracking

Object tracking is mainly the process of detecting an object in successive frames from a video. Tracking multiple objects in videos is an important problem in computer vision, applied in various video analysis scenarios, such as visual surveillance, sports analysis, robot navigation and autonomous driving. We have already talked about object detection, and one may wonder where are the differences between tracking and detection.

First of all, normally, tracking is faster, due to the fact that in each frame you have information about the object from the previous frames such as appearance, speed and direction. Second, tracking can handle some occlusions and preserve the identity of the object. Object detection can be a preceding step to object tracking, performed to check existence of objects in a frame and to precisely locate and classify the object.
Once the object is found, object tracking is used to follow the object through the consecutive frames. In each frame the movement of the object is computed by means of the motion model and the aspect by means of the appearance model. With the motion model we can predict a region that contains the object in the next frame, and with the appearance model we can use this region to find a more accurate position. As the appearance of the object can change drastically, a classifier that categorizes a region of the image as either an object or background is used. In image classification we have online and offline classifiers. Online classifiers are the ones trained at runtime, using very few examples at a time, while offline classifiers are trained using thousand of examples. The image classification used on trackers is created in an online manner, due to the fact that the results are needed at runtime.

Just like object detection, object tracking has different state of the art methodologies. While there exist different ideas studied under object tracking, such as optical flow, Kalman filter (which we will talk about in the next chapter) and meanshift and camshift, we will focus on those that can track a single object located first by an object detection algorithm. One of the first "trackers by detection", already obsolete, is the Boosting tracker [GGB06]. This tracker is based on an online version of Adaboost [FS97], the algorithm used in the Viola-Jones framework for detecting faces, trained at runtime with positive and negative examples of the object. For each frame, the classifier runs on every pixel in the neighborhood of the previous location, recording the maximum score where the location

of the object is. One of the problems of this tracker is that one does not know when the tracking has failed, as it always shows a location for the object. Another tracker is the Multiple Instance Learning (MIL) tracker [BYB09], based on the boosting tracker but instead of considering only the location of the object as a positive example when training, it looks in a neighborhood around the object and generates several positive examples. This tracker does not drift as much as the boosting tracker, and works well under partial occlusion, but still has the problem of the little reliability of the tracking failure reporting. Another tracker is the Kernelized Correlation Filters (KCF) tracker, based on [HCMB14], from the previous two trackers uses the fact that the multiple positive samples in the MIL tracker have large overlapping regions. This tracker is faster and more accurate than MIL and it reports tracking failure better, its only problem is that it does not recover from full occlusion. Now we have the Tracking, Learning and Detection (TLD) tracker from [KMM12], which, as the name says, tracks, learns and detects the object frame to frame. The detector localizes all appearances that have been observed so far and corrects the tracker, if necessary, and the learning estimates detector's errors and updates it to avoid them in the future. TLD works very well in occlusions over multiple frames and keeps track on scale changes, but it can have a lot of false positives. Finally, in 2015 the now state of the art tracker Clustering of Static-Adaptive Correspondences for Deformable Object Tracking (CMT), [NP15], appeared. Its main idea is to break down the object of interest into tiny parts (keypoints) and in each frame try to find these keypoints. First, they track the keypoints from the previous frame to the current frame by estimating its optical flow, and then they match the keypoints by comparing their descriptors. After this, every keypoint votes for the object center, so that the keypoints in the current frame that do not match the center are removed. The new bounding box is computed based on the remaining keypoints.

1.2 Quadcopter control

In the last few years there has been a growing interest in robotics, in concrete in Unmanned Aerial Vehicles (UAV). The advances in technologies like microcomputers and aerodynamics have made possible the appearance of small, low cost and easy to manage UAVs. Navigation is more challenging in flying robots than in ground robots, due to the fact that they require feedback to stabilize. For this reason, interest in tracking objects is increased when it is made with a flying robot. In particular, our focus will be on a small Vertical take-off and landing UAV, the quadcopter. Quadcopters have low dimensionality, good maneuverability, payload capability, and also a high energy consumption. Our need is for a low cost, easy to manage, that can be programmed quadcopter, and for this we chose the AR.Drone 2.0.

1.2.1 AR.Drone 2.0

A quadcopter is a multirotor helicopter with four actuators (propellers), each providing a force in the body-fixed z-direction and a torque to the body. The AR.Drone 2.0 is a quadcopter in x-configuration,

which means that the rotors are not aligned on the principal axes of the body-fixed coordinate-system, as in the +-configuration, see 1.9.

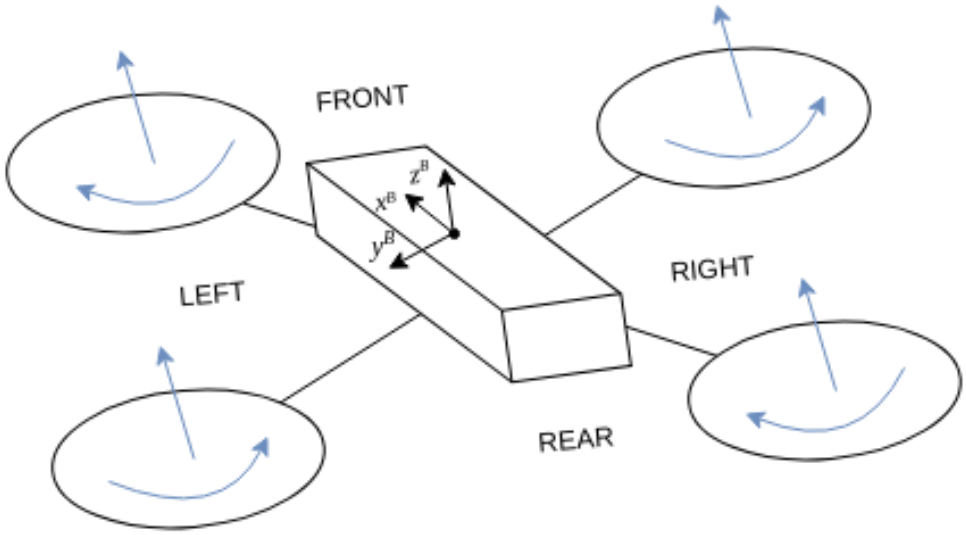


Figure 1.9: Quadcopter with x-configuration representation, with body-fixed coordinates frame *B*

Movements are obtained by changing the pitch, yaw and roll angles, and by changing the vertical speed, 1.10. To hover, all rotors may speed at the same velocity such that the global force of the quadcopter cancels the gravity force. For moving forward (backward) both front (rear) rotors have to decrease their velocity while the rear (front) ones increase them. The same applies for left/right moves and rotors.

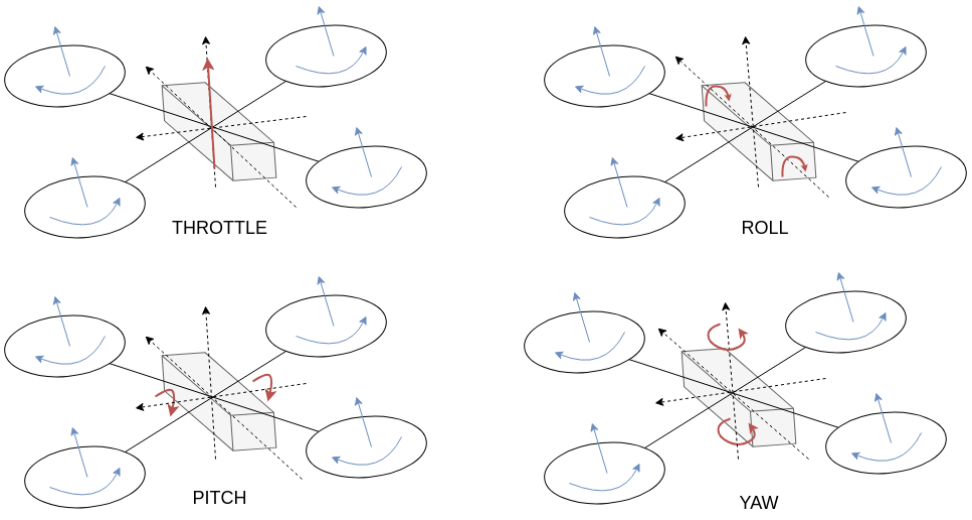


Figure 1.10: Roll, pitch, yaw and throttle movements

All quadrotors have two coordinate frames: the inertial frame (earth-fixed frame) and the base frame (body-fixed frame). This implies a system of six degrees of freedom (x, y, z, pitch, roll, yaw), controlled by adjusting the rotational speeds of the four rotors. With this, our system has four inputs

and six outputs, so some assumptions are made in order to control it: pretending the quadcopter to be a rigid body and the structure to be symmetric (no ground effect).

Without entering in depth in the explanation of the AR.Drone's hardware we will talk about the main sensors and actuators in the drone and how does its communication work. The engines have three current phases controlled by a micro-controller that makes sure all the engines work in coordination and stop if any obstacle is found. The drone uses 1000 mAh, 11.1V LiPo batteries to fly, and lands when it detects a low battery voltage. It has many motion sensors, located below the central hull: an inertial measurement unit, used for automatic pitch, roll and yaw stabilization and tilting control; an ultrasound telemeter, for automatic altitude stabilization and assisted vertical speed control; a camera aiming towards the ground, for automatic hovering and trimming; a 3-axis magnetometer and a pressure sensor. Finally, the AR.Drone 2.0 has a frontal camera, a CMOS sensor with a 90 degrees angle lens, that provides 360p or 720p image resolutions (also used for the ground camera), with a frame rate between 15Hz and 30Hz. The connection with the drone is made via a WiFi network:

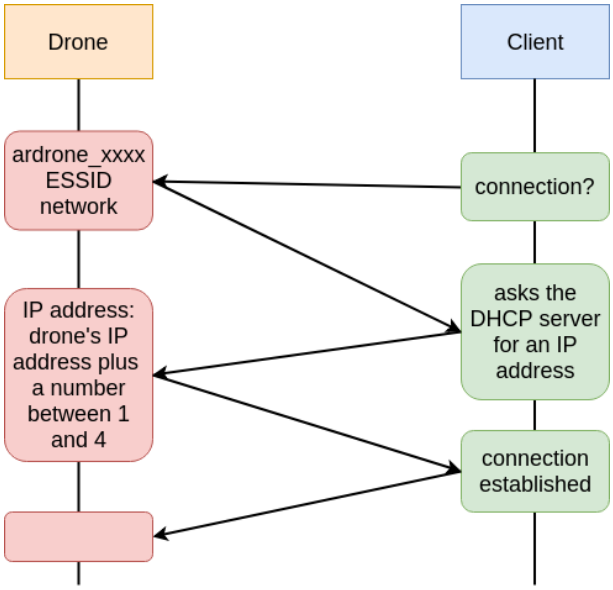


Figure 1.11: AR.Drone 2.0 Wifi connection

Once the connection is established we have 4 main services for communication:

- The control and configuration of the drone is done by sending AT commands on UDP port 5556. These commands are to be sent on a regular basis (30 times per second).
- Information about the drone (status, position, speed, etc.) - navdata - is sent on UDP port 5554.
- Video stream is sent by the drone to the client device on TCP port 5555.

- A fourth communication channel (control port) can be established on TCP port 5559 to transfer critical data.

1.2.2 Available open-source libraries

Any client device supporting WiFi can control the AR.Drone 2.0. But what are the commands that the drone understands? And how can we make it fly?
Parrot created a documentation [PBED12] that explains the format of the text strings that can be sent to the drone to control its actions (AT commands). According to this documentation: *These strings are encoded as 8-bit ASCII characters with a Carriage Return character as a new line delimiter. One command consists in the three characters AT* followed by a command name, an equal sign, a sequence number, and optionally a list of comma separated arguments. An AT command must reside in a single UDP packet, and the maximum length of the total command can't exceed 1024 characters.* With this documentation and these AT commands, we could be able to create our own framework for controlling the drone, but that would take time, and in this project our main objective is not to create a framework to control the drone, but to be able to make the drone follow a person. There exist open-source libraries that can ease our work:

- Parrot also created an SDK that simplifies the work of writing an application to remotely control the drone. This SDK is divided into two libraries: ARDroneLIB and ARDroneTool. With these two libraries the work of controlling the drone is much more easier. But it is still hard to understand, the documentation is outdated and the few examples on Linux only work on 32-bit computers.

- Another famously used library for controlling the AR.Drone 2.0 is *ardrone_autonomy*, a ROS driver based on the Parrot SDK. ROS (Robot Operating System) is a set of utilities and libraries for implementing all different kinds of functionality on robots. The problem of using ROS is that some time is needed to get familiar with its structure.

- If you are familiar with node.js the node-ar-drone is your module. NodeJS is a popular plugin-based JavaScript server platform, which runs locally.

- For Python we have several open-source libraries, but they have basically the same format: a python document where several functions that create AT commands sent to the quadcopter are written, and with some low-level functions that manage the navdata received from the quadcopter. PS-Drone is one of these libraries, it is designed to also run on really slow computers, has a blog where bugs or question can be asked and are pretty quickly answered, and there exists a documentation with all the functions explained and some easy examples shown.

2 Framework overview

In the previous chapter we discussed the different available techniques that can be used for the purpose of controlling a quadcopter by following a person with the use of computer vision, in particular object detection and object tracking, and introduced the AR.Drone 2.0 hardware specifications together with a series of open-source libraries created for the quadcopter's easy control. In this chapter we use the object detection and tracking methods from sections 1.1.2 and 1.1.3 to create a software able to keep track of a person through the quadcopter's movement, recover from tracking failure, and move the quadcopter accordingly to the person's movements, with the use of one of the open-source libraries introduced in 1.2.2 to easily send the commands to the drone.

In figure 2.1 a flux diagram of our implementation is shown, with the introduction of some new elements of which we will talk about in the next sections, like the Kalman filter or the PID algorithm.

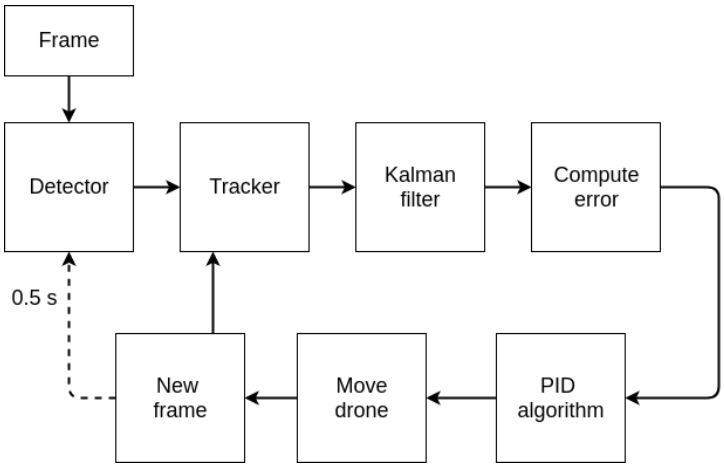


Figure 2.1: Schematic representation of the software structure.

The AR.Drone 2.0 sends every few milliseconds a frame captured from its camera to our device, and with the processing of each of these frames we will be able to send the commands to the drone that will make it go to a certain position. To simplify, we fix this position to be, in each frame:

- x so that the person's bounding box horizontal center corresponds to the horizontal center of our frame.
- y so that the person's bounding box vertical center corresponds to the vertical center of our frame.

- z so that the person's bounding box height is close to the person's bounding box height at the starting frame.

With these requirements a quadcopter controller is created, to find the best combination between the drone's stability and movement.

In the next sections we will thoroughly explain the different approaches used in each step of the implementation.

2.1 Detection and tracking of the person

Once the drone takes off, and after a few seconds to let it stabilize, the processing of the frame starts and an object detector is used to find persons in the image. As no interaction with the program is needed, it is required that only the person to be followed stands in front of the drone, otherwise it could detect another person to track. The object detector used is based on the SSD approach and has as feature extractor the MobileNet architecture [LAE+15, HZC+17]. This approach was chosen for its high speed (it can be computed in a smartphone CPU), and its great accuracy given the circumstances.

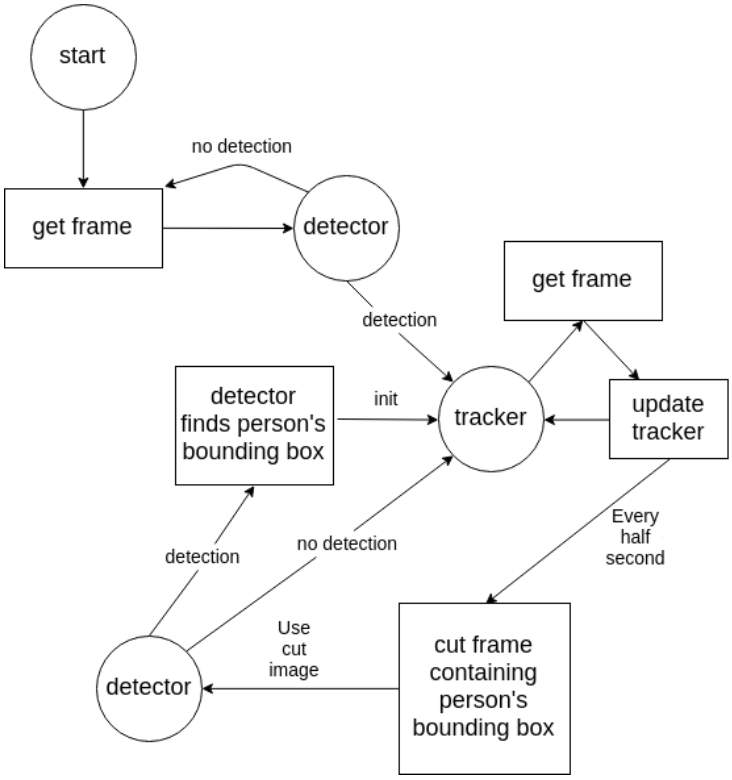


Figure 2.2: Object detection and tracking implementation.

The object detector is used to find the person in the first frames. Once detected, the corresponding bounding box is used to keep the detection through the following frames with an object tracker. In section 1.1.3 we explained several object trackers and introduced the state of the art: CMT [NP15], but the software used in the detection, explained in 3.2.2, forces us to use the OpenCV implementation of KCF [HCMB14]. This tracker is able to obtain the x and y center positions of our person in the frame. This is, in each frame the initial bounding box is moved through the image to wrap around the person. But this bounding box does not change in area, it always has the same width and height, which makes impossible to know the distance between the drone and the person by only using the tracker. To solve this, a combination between our detector and our tracker has been developed in this project, to try to find the best balance between speed and the accuracy of the person's bounding box. In figure 2.2 we observe a schematic representation of how this balance is obtained. The distance between the quadcopter and the person is updated every half second, also preventing the tracker from drifting, by using the object detector with only a part of the image that contains the person. With each of these new detections a new tracker is created, initializing it with the bounding box obtained by the detector.

2.2 Kalman filter and error computation

The AR.Drone 2.0 is not the best quadcopter in the market, and when flying or hovering there exists a trembling in the quadcopter that entails a trembling in the frames. Also, the tracker used does not produce a fluid trajectory of the detected bounding box between frames. This noise can be reduced with the help of the Kalman filter. The Kalman filter is an algorithm that uses a series of observations over time and a predictive model, considering that none of them are perfect, and tries to find the best balance between them, creating a detection that is more accurate that with only using each observation alone.

Next we will explain the setup from which we will create our Kalman filter:

Let x_k be the state vector that describes our person's position and velocity at detection k

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{bmatrix}$$

where (x, y) is the position in pixels of the person in the frame, and $(\dot{x}, \dot{y})$ is the velocity (the derivative of position with respect to time).

The model

We assume now that between the detections $k-1$ and k there is a constant stochastic acceleration $\mathbf{a}_k = [a_x^k, a_y^k]$ normally distributed with mean 0 and standard deviation $\sigma_{\mathbf{a}}$. We know, from kinematics, that with initial state $[x_0, y_0, \dot{x}_0, \dot{y}_0]$, we have:

$$\begin{cases} \dot{x}(t) = \dot{x}_0 + a_x t \\ \dot{y}(t) = \dot{y}_0 + a_y t \end{cases}$$

so that

$$\begin{cases} x(t) = x_0 + \dot{x}_0 t + \frac{1}{2} a_x t^2 \\ y(t) = y_0 + \dot{y}_0 t + \frac{1}{2} a_y t^2 \end{cases}.$$

From this, we observe that our state in detection k is

$$\begin{cases} x_k = x_{k-1} + \dot{x}_{k-1}\Delta t_k + \frac{1}{2} a_x^k \Delta t^2 \\ y_k = y_{k-1} + \dot{y}_{k-1}\Delta t_k + \frac{1}{2} a_y^k \Delta t^2 \\ \dot{x}_k = \dot{x}_{k-1} + a_x^k \Delta t_k \\ \dot{y}_k = \dot{y}_{k-1} + a_y^k \Delta t_k \end{cases}$$

which concludes that

$$\begin{pmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta t_k & 0 \\ 0 & 1 & 0 & \Delta t_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{k-1} \\ y_{k-1} \\ \dot{x}_{k-1} \\ \dot{y}_{k-1} \end{pmatrix} + \begin{pmatrix} \frac{1}{2}\Delta t_k^2 & 0 \\ 0 & \frac{1}{2}\Delta t_k^2 \\ \Delta t_k & 0 \\ 0 & \Delta t_k \end{pmatrix} \begin{pmatrix} a_x^k \\ a_y^k \end{pmatrix}$$

that can be expressed as

$$\mathbf{x}_k = \mathbf{F}_k \mathbf{x}_{k-1} + \mathbf{G}_k \mathbf{a}_k$$

with

$$\mathbf{F}_k = \begin{pmatrix} 1 & 0 & \Delta t_k & 0 \\ 0 & 1 & 0 & \Delta t_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathbf{G}_k = \begin{pmatrix} \frac{1}{2}\Delta t_k^2 & 0 \\ 0 & \frac{1}{2}\Delta t_k^2 \\ \Delta t_k & 0 \\ 0 & \Delta t_k \end{pmatrix}.$$

Setting $\mathbf{w}_k = \mathbf{G_k}\mathbf{a}_k$ and knowing that $\mathbf{a}_k$ is a stochastic variable we can assume that $\mathbf{w}_k \sim N(0, \mathbf{Q}_k)$ is the zero mean Gaussian distributed process noise, with $\mathbf{Q}_k$ being the covariance matrix from time

step $k-1$ to time step k:

$$\mathbf{Q}_k = \mathbf{G}_k \mathbf{G}_k^T \sigma_{\mathbf{a}}^2 = \begin{pmatrix} \frac{1}{4}\Delta t_k^4 \sigma_{a_x}^2 & 0 & \frac{1}{2}\Delta t_k^3 \sigma_{a_x}^2 & 0 \\ 0 & \frac{1}{4}\Delta t_k^4 \sigma_{a_y}^2 & 0 & \frac{1}{2}\Delta t_k^3 \sigma_{a_y}^2 \\ \frac{1}{2}\Delta t_k^3 \sigma_{a_x}^2 & 0 & \Delta t_k \sigma_{a_x}^2 & 0 \\ 0 & \frac{1}{2}\Delta t_k^3 \sigma_{a_y}^2 & 0 & \Delta t_k \sigma_{a_y}^2 \end{pmatrix}$$

The observations

At each time step k a noisy measurement $\mathbf{z}_k$ of the true position of the person is made. Let us suppose this noise $\mathbf{v}_k$ is also normally distributed with mean 0 and standard deviation $\sigma_z = [\sigma_{z_x} \ \sigma_{z_y}]$:

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k$$

As our observation is of the person's position, and not its velocity, we have that $\mathbf{H}_k = [1\ 1\ 0\ 0]$ and the covariance matrix of the observation is

$$\mathbf{R}_k = \begin{pmatrix} \sigma_{z_x}^2 & 0 \\ 0 & \sigma_{z_y}^2 \end{pmatrix},$$

for all time steps k.

Initial state

The filter is not initialized until an observation is made. The initial state is composed by the first observation for the position and $[0,0]$ for the velocity. As this initial state is not know perfectly, we set as initial covariance matrix

$$\mathbf{P} = \begin{pmatrix} \sigma_x^2 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 \\ 0 & 0 & \sigma_{\dot{x}}^2 & 0 \\ 0 & 0 & 0 & \sigma_{\dot{y}}^2 \end{pmatrix}.$$

After preparing the settings needed to create the filter, we can explain how does it work and how is the previous setup used for the computation of an estimated position.

The Kalman filter is a recursive estimator, which means that only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state. At every time step the algorithm has two stages:

- The **Predict** step. Uses the estimated state from the previous time step to produce an estimate at the current time step. This predicted state is also know as the *a priori* state estimate because no measurement information has been incorporated in the estimation.
- The **Update** step. The current *a priori* prediction is combined with the current observation to refine the state estimate. The updated step is also known as the *a posteriori* state estimate.

The state of the filter is represented by two variables: the $\hat{\mathbf{x}}_{m|n}$ is the state estimate at time step m given $n \leq m$ measurements, and the $\mathbf{P_{m|n}}$ is the error covariance matrix at state estimate $\hat{\mathbf{x}}_{m|n}$. The equations of the Kalman filter for the predicted step are

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} \tag{2.1}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{Q}_k \tag{2.2}$$

and for the updated step are

$$\mathbf{S}_k = \mathbf{R}_k + \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T \tag{2.3}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \mathbf{S}_k^{-1} \tag{2.4}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1}) \tag{2.5}$$

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\mathbf{P}_{k|k-1}(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)^T + \mathbf{K}_k \mathbf{R}_k \mathbf{K}_k^T \tag{2.6}$$

where $\mathbf{S}_k$ is the innovation covariance, $\mathbf{K}_k$ the *Optimal* Kalman gain, $\hat{\mathbf{x}}_{k|k}$ the *a posteriori* state estimate, and $\mathbf{P}_{k|k}$ the *a posteriori* state estimate covariance.

At each iteration the tracker updates the position of the person, and this position is improved by using these equations of the Kalman filter. The filtered position returned by the Kalman filter is the *a posteriori* state estimate computed in (2.5).

As we explained in the introduction of this chapter, the quadcopter's 3D desired position depends on the bounding box created by the tracker. With the Kalman filter, the (x, y) center point of the bounding box has been improved, but our objective is to minimize the difference between the center of the AR.Drone captured frame and this filtered position. Moreover, the quadcopter distance from the person has to be computed and controlled, we have to obtain the distance between the person and the quadcopter at startup and try to maintain this same distance during all the iterations. For this reason, we use the height of the bounding box at the first detection and compare it with the actual height. These differences compose an error that we will try to minimize by sending the correct commands to the drone. This 3-dimensional error is

$$\begin{cases} e_x = f_x - \frac{w_f}{2} \\ e_y = \frac{h_f}{2} - f_y \\ e_z = h_k - h_0 \end{cases}$$

where f_x, f_y is the filtered position, w_f, h_f is the resolution of the quadcopter's front camera, h_0 is the height of the person's bounding box at the first detection and h_k is the height of the person's bounding box at the actual iteration.
Knowing that in the roll movement the quadcopter goes right when the e_x error is positive, to go right the horizontal center of the frame has to be smaller than the x coordinate of the observation, and to go left it has to be bigger. However, OpenCV reads images vertical axis from top to bottom,

for this reason, and knowing that the throttle movements goes up when the e_y error is positive, the quadcopter goes up when the vertical center of the frame is bigger than the y coordinate of the observation, and goes down when it is smaller.

2.3 PID controller

Once the position of the person is obtained, its noise is removed with the Kalman filter and the error between person and quadcopter is computed, we can move the quadcopter to correct this error. But to do this we need a controller that helps us create a fluid trajectory, to avoid changes in the orientation at every frame. In this section we will explain the controller used, called Proportional-Integral-Derivative or PID controller.

The PID controller is the most common control algorithm used in the industry of automation. It is a control loop feedback mechanism which computes the deviation between a given value (measured process value) and a desired value (set point) and corrects it based on the proportional, derivative and integral terms (giving the controller its name). In figure 2.3 we have a block diagram that shows how a PID works.

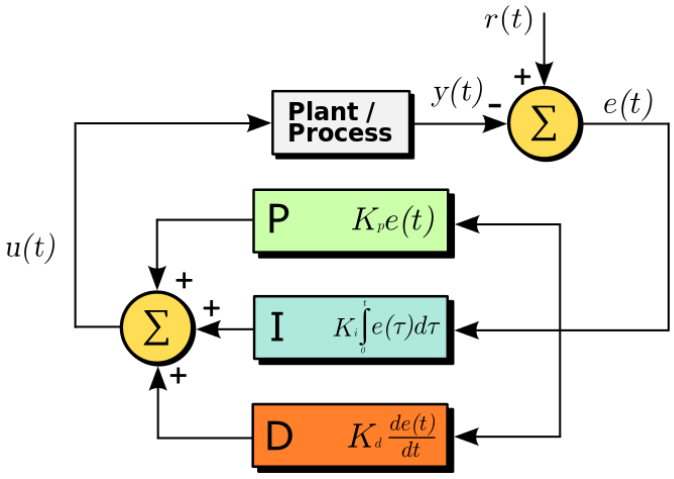


Figure 2.3: PID algorithm [Com16b].

The $r(t)$ value represents the set point and the $y(t)$ value the measured process value. The $u(t)$ value is the control signal and is described by the sum of the three terms: the **P**-term (proportional to the error), the **I**-term (proportional to the integral of the error), and the **D**-term (proportional to the derivative of the error). The equation of the PID controller is described by:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{2.7}$$

The controller has to be tuned in order to suit the dynamics of the process to be controlled. Giving the controller the wrong K_p, K_i and K_d parameters will lead to instability and slow control performances. There exist different types of tuning methods that can be used:

- **Trial and Error method**. With this simple method first we have to start with both $K_i$and K_d parameters to zero and increase K_p until the system reaches an oscillating behaviour. Then, we adjust the K_i parameter to stop the oscillation and finally adjut K_d for faster response.

- **Process reaction curve technique**. This method produces response when a step input is applied to the system. At first we apply some control output to reach a steady state (close one), then, in open loop, we generate a small disturbance and the reaction of the process value is recorded. This process curve is then used to calculate the $K's$ parameters. The method is performed in open loop so no control action occurs and the process response can be isolated.

- **Zeigler-Nichols method [ZN93]**. In this method, as in trial and error, the K_i and K_d parameters start at zero. The proportional gain is increased until reached the ultimate gain, K_u, at which the output of the loop starts to oscillate. The term K_u and the oscillation period T_u are used to set the gains as showed in figure 2.4.

Controller	Kp	Ki	Kd
P	0.5Ku		
PI	0.45Ku	0.54Ku/Tu	
PID	0.6Ku	1.2Ku/Tu	3KuTu/40

Figure 2.4: Table showing the Ziegler-Nichols method

But to tune a quadcopter, and without entering into more complicated algorithms, the best of these tuning methods is Trial and Error. Also, although for other systems the integral term is adjusted before the derivative one, in quadcopters is best to tune first the derivative one, and the integral has to be very small to avoid oscillations.

2.4 Drone movement

As explained in section 1.2.1, the AR.Drone 2.0 has four movements: roll, pitch, yaw and throttle. To control the quadcopter, a PID controller has to be created for each of these movements. Because of this, in this project we will have four controllers, each of which has to be tuned independently to find the correct parameters that will led to a smooth movement in its direction. The **roll** movement is modified by looking at the **x**-error filtered with the Kalman filter. After being computed, this error is then passed to the x-PID controller, which returns the velocity that has to be passed to the quadcopter. Equally, the **throttle** movement is modified by looking at the **y**-error filtered with the Kalman filter and then computing its velocity with the **y**-PID controller. For the **pitch** movement the velocities are obtained with the **z** component, when comparing, every half second, the difference between the original height of the person and the actual height. Finally, we have the **yaw** movement, that we did not implement for this project.

In section 1.2.2 we introduced different ways of sending to the quadcopter the commands needed to produce these movements. Although a few of them are valid to meet with our objectives, we chose the one that uses Python as programming language: the *PS-Drone* API. In the next chapter we will explain how this library is structured and how the commands are sent to the quadcopter.

3 Software implementation

In this chapter we will explain how the theoretical framework from the previous chapter has been implemented.

3.1 Source code organization

The software created for this project is written in Python 2.7 and uses OpenCV 3.4.0 to obtain the pre-trained model for the detection. The computer used is a Dell XPS 13 with an Intel(R) Core(TM) i7-7560U CPU @ 2.40GHz. All the computations, including the detection of the person with a neural network, are made in this computer. All the source code can be found in the Github repository[1]

Figure 3.1 shows the project's class diagram with the structure of the system.

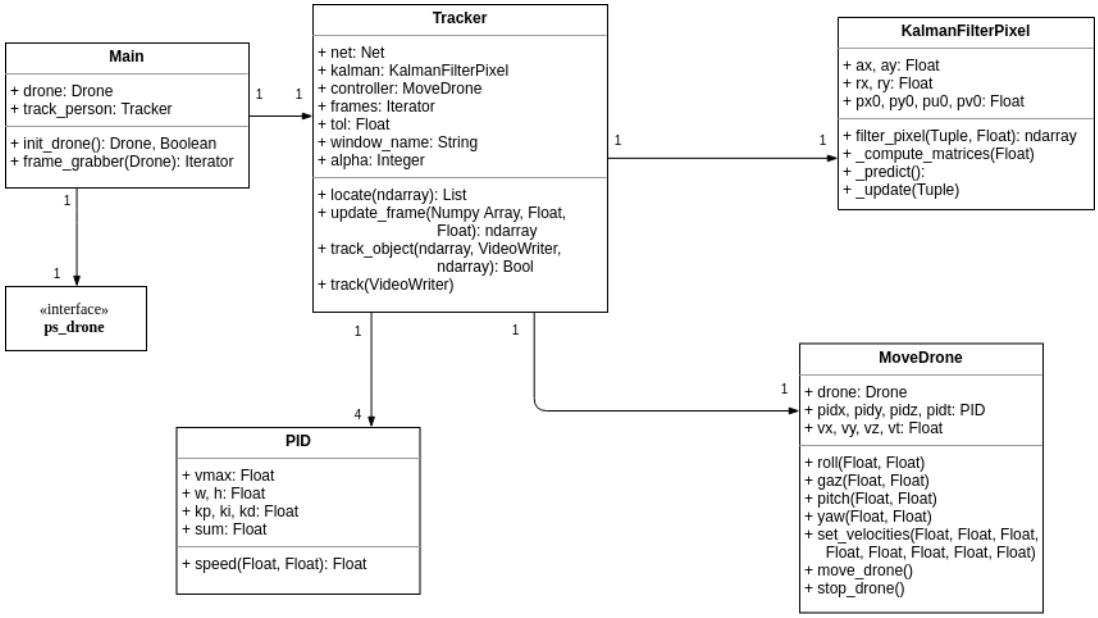


Figure 3.1: Class diagram of the implemenation.

The program is composed of the following files:

- *main.py* The main file initializes the quadcopter with the PS-Drone API, and creates a frame grabber that is passed to the tracker. The PID parameters are initialized here and the network used for the detection is extracted from a file in the containing folder.

[1] https://github.com/juliiaa28/ARPET.git

- *tracker.py* This is the most important file of the program. For every frame, the detector and the tracker work together to keep the bounding box of the person, then the Kalman filter obtains an improved position and the controller moves the drone. Finally, the parameters for the next iteration are updated.
- *kalman_filter.py* Implements the equations explained in section 2.2.
- *pid.py* This file contains a class to modify and set the parameters of the pid controller.
- *ps_drone.py* This is an external file and contains the PS-Drone API retrieved from www.playsheep.de/drone.
- *move_drone.py* This file contains all the necessary commands to move the drone. It also contains functionalities to change the desired velocities that will be used by the PID controller.

3.2 Software explanation in depth

In this section we will explain how the classes from figure 3.1 are implemented.

3.2.1 Initialization

In figure 3.2 we have a flux diagram of the drone startup. Once the program is started, the first thing to do is check whether the quadcopter has enough battery to take off or not. With low battery the AR.Drone 2.0 can show video images but does not take off.

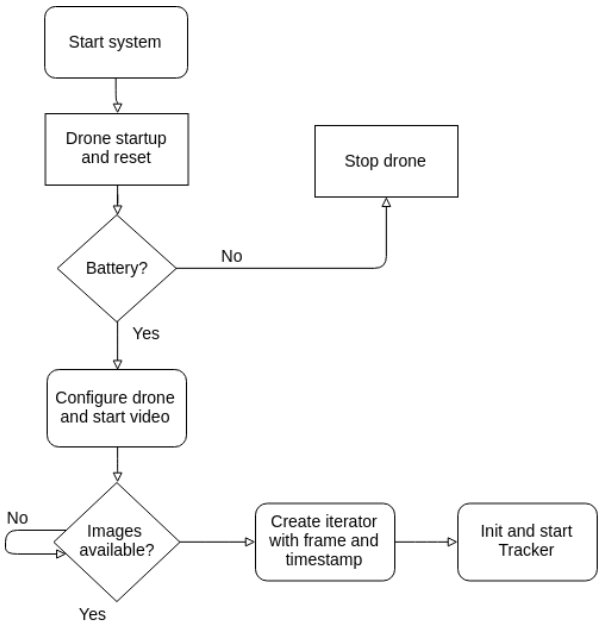


Figure 3.2: Flux diagram of the startup implementation

3.2.2 Tracking algorithm

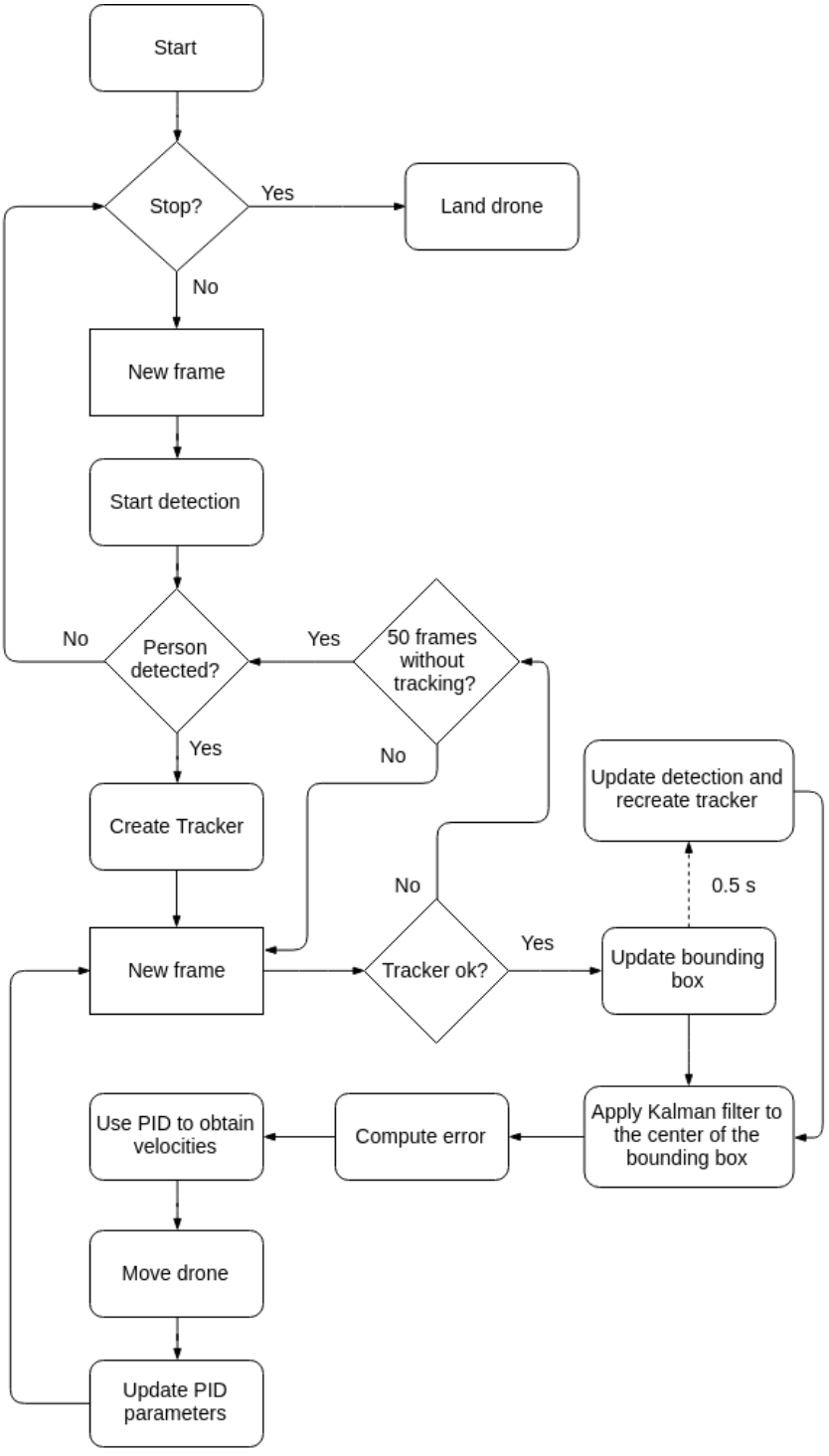


Figure 3.3: Flux diagram of the tracking implemenation.

In figure 3.3 we have a flux diagram showing how the tracking algorithm is used in the software, after the startup of the system (showed in diagram 3.2), and followed by the correction of the position with the Kalman filter and the move of the quadcopter using the PID controller. The tracking keeps in

the loop until the user stops the quadcopter by clicking the *Esc* keyboard button, which automatically makes the drone land.

Detection and tracking configuration

As explained in section 2.1, both detector 1.1.2 and tracker 1.1.3 are combined to obtain a robust tracking of the person. The detector is created using the *dnn* module of the OpenCV library. In paticular, a Caffe [JSD+14] pre-trained model is used to create the object detector, which is a version of the original Tensorflow implementation from [AAB+16]. This pre-trained model allows us to use an implementation of the MobileNet-SSD network, trained with datasets such as COCO and PASCAL VOC, without wasting time doing it ourselves, and obtaining a model that detects 20 objects in images (dogs, cats, people, sofas, etc.). To use a pre-trained Caffe model with the dnn module of OpenCV we need two files that can be retrieved from the Caffe website:

- A Caffe prototxt file that defines the structure of the neural network.
- A binary .caffemodel file that includes the pre-trained model.

Now that we understand how the network is created and used to detect the person at each frame, let's explain how the tracker works. OpenCV 3.0 comes with a tracking API with 6 different trackers, some of which were explained in 1.1.3. Our choice was the KCF tracker which has the best accuracy, although it does not recover from full occlusion. The tracker is initialized using the bounding box retrieved from the detector and it is updated with every frame.

But, as explained in section 2.1, the tracker update does not change the dimensions of the bounding box, so the distance from the quadcopter to the person cannot be computed with only this solution. That is why we created a combination between detection and tracking showed in figure 2.2 , which finds a balance between speed and accuracy.

Filtering the position

In section 2.2 we explained in depth the equations used to create the Kalman filter for this particular case. Basically, we assumed a constant velocity model with an acceleration of the pixel with a diagonal covariance. The Kalman filter class is initialized with 8 parameters:

- **ax, ay**: The standard deviation of the pixel acceleration (x,y respectively).
- **rx, ry**: The standard deviation of the pixel position observation (x,y respectively).
- **px0, py0**: The standard deviation of the pixel initial position (x,y respectively).
- **pv0, pu0**: The standard deviation of the pixel initial velocity (x,y respectively).

The initial state vector $\mathbf{x}_{0|0}$(position and velocity) is assumed to be zero, and with the ($px0, py0$, $pv0, pu0$) parameters the initial state covariance matrix $\mathbf{P}_{0|0}$ can be created.

At each iteration k the Kalman algorithm follows the steps observed in the diagram from figure 3.4 which is based on the equations and matrices explained in section 2.2. The new observation and the time of the observation are passed to the filter. Using the difference between the time of the last observation and this observation time, the $\mathbf{F}, \mathbf{H}, \mathbf{Q}, \mathbf{R}$ matrices are computed. Then, the linear model prediction is fused with the observation using the Predict-Update steps, and the new state is returned.

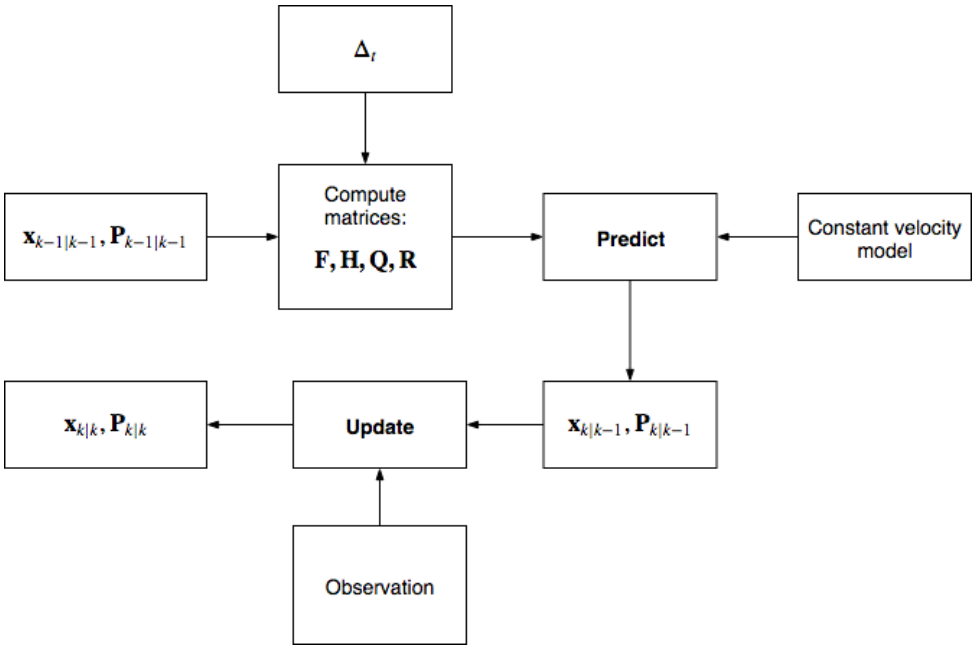


Figure 3.4: Flux diagram of the Kalman filter

PS-Drone

Before talking about the PID controller and how the velocities are computed, let's first explain how the used library works, and which parameters it needs in order to send the correct commands to the quadcopter, that make it move as desired.

PS-Drone is created in a single file, called *ps_drone.py*, and has a complete documentation found in www.playsheep.de/drone. For this project we only use the functions to configure the drone, to obtain video images, and to move the drone in the desired direction. Here is the list of functions used:

- *startup()*: to connect to the drone.
- *reset()*: initiates a soft reset of the drone.

- *trim()*: the drone sets the reference on the horizontal plane.
- *getSelfRotation()*: the drone measures out the yaw gyrometers self spinning.
- *hdVideo()*: sets the drone's video stream to H.264 encoded, with an image resolution of 1280 × 720.
- *frontCam()*: switchs to the drone's front camera.
- *startVideo()*: activates and processes drone's video, images are available in the *VideoImage* attribute.
- *getBattery()*.
- *VideoImageCount*: sequential number of the decoded video images stored in *VideoImage*.
- *VideoImage*: contains the actual video-image of the drone as an OpenCV2 image-type, when video is activated.
- *VideoDecodeTimeStamp*: time when the video image in *VideoImage* was decoded.
- *VideoDecodeTime*: time needed to decode the video image in *VideoImage*.
- *move()*: drone moves to all given directions in given speed. The usage is as follows: move(roll, pitch, throttle, yaw). This is:
  - roll: a float value from -1 to 1, where -1 is full speed to the left and 1 full speed to the right. This value, called the ϕ-angle, is a percentage of the maximum inclination configured for the left-right tilt.
  - pitch: a float value from -1 to 1, where -1 is full speed backward and 1 is full speed forward. This value, called the θ-angle, is a percentage of the maximum inclination configured for the front-back tilt.
  - throttle: a float value from -1 to 1, where -1 is full speed descent and 1 is full speed ascent. This value, called *gaz*, is a percentage of the maximum vertical speed.
  - yaw: a float value from -1 to 1, where -1 is full speed left spin and 1 is full speed right spin. This value, called ω, is a percentage of the maximum angular speed.
- *stop()*: the drone stops all movements and holds position. Note: Setting the *move()* parameters all to 0 would not stop movement, but stop the acceleration. To stop movement this function has to be used.

The *startup()* function creates a socket that connects to the quadcopter's IP, which is 192.168.1.1 as default, and sends the four initial commands to the drone. Then two processes for the VideoData and the NavData (sensor data) configuration are created. These two processes send to the quadcopter the configuration data needed to initiate the communication through the video data and the navdata

ports. The AR.Drone 2.0 sends its NavData at UDP port 5554, which means that the quadcopter keeps on sending without caring if the data is being received, and its VideoData at TCP port 5555, making sure that all the frames arrive safely.
After this, two threads are created: the first is used to receive data from the drone and the second to send data to the drone. The first thread checks the ip of the received pipe and compares it with the NavData pipe and the VideoData pipe, depending of which receives the sensor values from the NavData process or the image data and feedback of the Video process. The second thread is used for sending the configuration to the quadcopter. It is asynchronous but secure: the configuration requests are in a queue and, after sending the first entry, the thread waits for the next NavData package to be received and then checks whether the configuration has been correctly set, if not, the value is requeued. With this the configuration data is always received by the quadcopter, but not always in the initial order.

The commands used to control the drone are sent inside UDP packets on port UDP-5556. These are low level commands which are created from the basic drone commands (from functions like *takeoff()*, *move()* or *stop()*) using the *at()* function, which receives a word (for example **REF** for taking off or landing, or **PCMD** for moving the drone) and some parameters such as the *roll*, *pitch*, *yaw* and *throttle* velocities, and creates another command that the drone is capable of understanding. These commands are not sent within a thread but in the main process, for they have to be received and implemented instantly. The commands are to be sent every 30 ms for smooth drone movement, if the drone does not receive two consecutive commands within less than 2 seconds it can consider the WiFi connection as lost. The port used is UDP, which means that some commands may be lost, but as the messages are constantly sent, one lost command is not a problem.

Move the drone with the PID controllers

Now that we know the parameters needed to move the drone, we can create the controllers in order to obtain the velocities (or movement percentages) in the required range [-1,1]. If we look at the equation of a PID controller, 2.7, we see that, as it is a linear combination of terms, the output is not comprised in our desired range. Because of this, we use the *hyperbolic tangent* function:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

In figure 3.5 we have a plot of the hyperbolic tangent function, showing that the output is comprised in the desired range [-1,1].
The equation used for each of the PID controllers is then:

$$pid(t) = tanh\left(K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}\right) \tag{3.1}$$

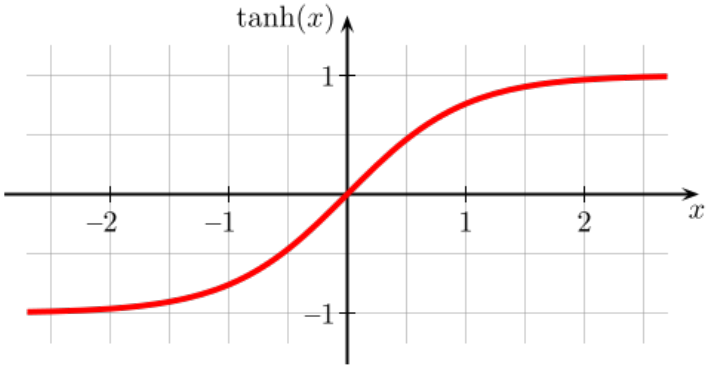


Figure 3.5: Hyperbolic tangent plot from [Com17]

This equation has to be discretized in order to be used in the controller. The discretized PID equation at iteration k is:

$$pid_k = tanh\left(K_p e_k + K_i \sum_k e_k + K_d(e_k - e_{k-1})\right) \tag{3.2}$$

The output of each of the controllers is then passed as a parameter in the *move()* function of the PS-Drone library.

4 Results

In this section we will evaluate each part of our application. First, we will start by testing the created algorithm for detecting and tracking a person, observing how the algorithm reacts at different occlusions or loss of target. Second, we will apply the Kalman filter and observe if the trajectory of the quadcopter in the x and y coordinates is indeed smoother. Finally, we will check if, with the tuned PID controller parameters, the quadcopter is capable of following a person.

4.1 Tracking robustness

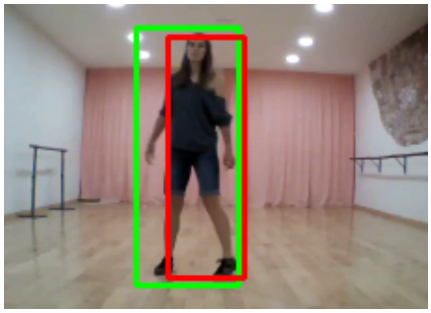
This part of the project depends on the object detector (SSD-MobileNet) and the object tracker (KCF) algorithm explained in 2.1. It was tested in two different parts. First, the system is tested with the quadcopter landed, observing how the detection and tracker behave in a still camera. Second, the test is equally done, but with the drone taken off and moving. This second test is done after the whole system is created (Kalman filter and PID controller), but we will only test how both detector and tracker work and not the quadcopter's movement.

4.1.1 Landed quadcopter

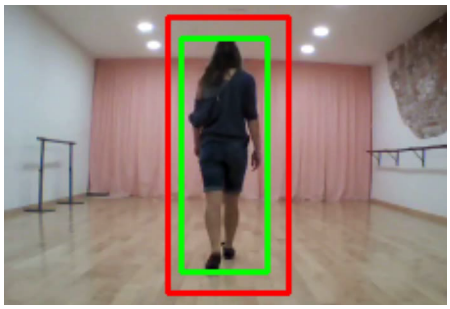
The system with the landed quadcopter behaves in a clearly good way when losing the target. The KCF tracker can sometimes lose the person, and create a tracking failure, but the designed algorithm uses the object detector in the whole image to find again the person, and the tracking continues. We must say that if the target is moving too fast the tracker tends to lose the person, and that if in a tracking failure another person appears in the image (or detects a wrong person) it can begin to track this person or object.

In figure 4.1 a series of consecutive images is shown. In the first frame the person is detected by the tracker, in the second frame a tracking failure occurs, which is recovered later on as shown in the third frame.

Figure 4.1: Consecutive frames showing a tracking failure recovery.



(a) A person moving to the left.



(b) A person going backwards.

Figure 4.2: Detection (green rectangles) and tracking (red rectangles) in two different frames.

In figure 4.2 we can see two frames with two bounding boxes: a green one and a red one. The green rectangle is the detection created every half second by the object detector, while the red rectangle is the tracker. The detector creates another bounding box, corrected from the tracking one, with which a new tracker will be created. In frame 4.2a the green bounding box grabs the whole person, while the red one is smaller and briefly drifted to the right. In frame 4.2b the red bounding box is much more bigger than the person, while the green one is smaller and fitting the person better.
Let's try how the tracker behaves when another person is in the field of vision. In figure 4.3 we observe that the targeted person is correctly tracked while another person is walking by its side. In this test the tracker correctly kept following the target.

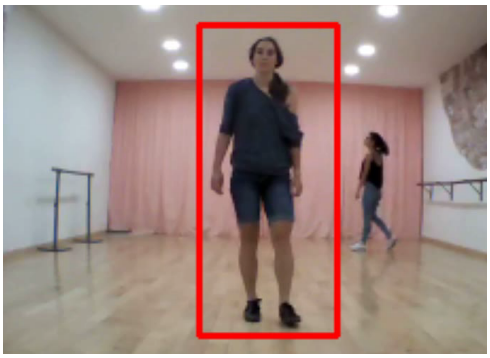


Figure 4.3: Tracking algorithm correctly following the target while another person is passing.

In figure 4.4 we have three consecutive frames showing how a person walks between the quadcopter's camera and the target. While in the middle frame the tracker lost the target, once the person passes and the target is again visible the KCF tracker is capable of keeping the correct track. This is because when a tracking failure occurs, the tracker keeps looking for the person during 50 frames, if after these frames the person is not found, the detection starts looking for another person at the whole image. In this case, the tracker has time to recover from occlusion and to find again the person.

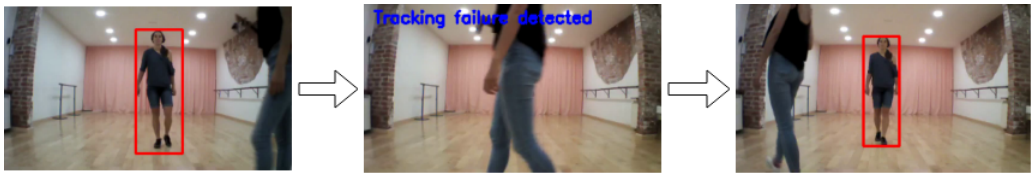


Figure 4.4: Consecutive frames showing a second person occluding the target.

Finally, we will show the two errors explained before: the loss of a too fast moving target and the wrong detection when the target is lost. In figure 4.5 we have four consecutive frames that show how these errors occur. In the first frame the target is correctly tracked, but when it keeps running the tracker loses it, as shown in the second frame. In the third frame the detector is trying to find a person to track, but wrongly thinks that the column is a person as shown in the fourth frame.

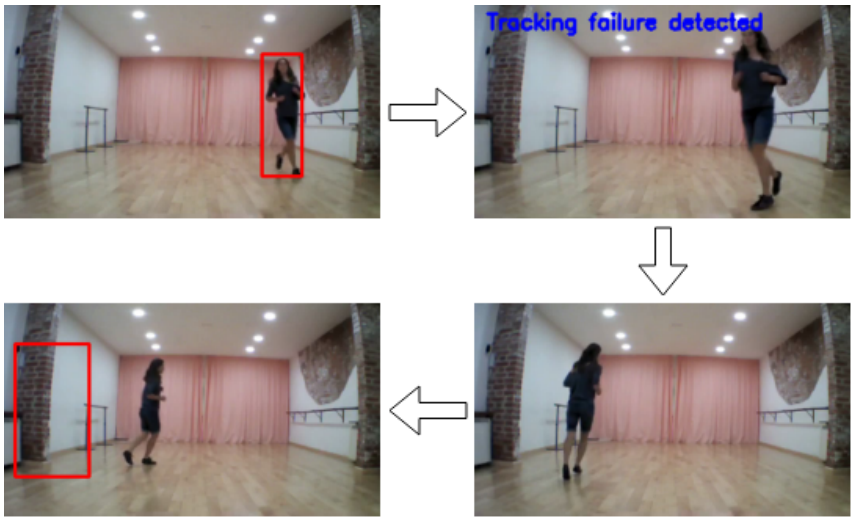


Figure 4.5: Consecutive frames showing the loss of the target and the wrong detection.

4.1.2 Quadcopter in motion

The tests with the quadcopter flying showed more tracking failures than with the landed quadcopter. This is because the movement of the quadcopter produces a faster movement of the target, and knowing, as shown in figure 4.5, that the tracker fails more as the target moves faster, implies that, with a moving quadcopter and person, the tracker performance is worse.

From now on, all the images shown in this section are taken with the quadcopter flying. We will follow the same tests as in the previous subsection, and compare the tracking observed in both videos.

In figure 4.6 we observe, like in figure 4.1, three consecutive frames showing a tracking failure due to occlusion, but in this case the quadcopter is moving. We observe that the quadcopter is sufficiently stable, because when the target is lost the quadcopter is told to stop. The number of seconds between

the first frame and the third is eight.

Figure 4.6: Consecutive frames showing a tracking failure recovery with a moving quadcopter.

In figure 4.7 we have, as in figure 4.2, the two rectangles showing both detection and tracking frames. The adjustment of the bounding box when the quadcopter is moving works as well as when it is landed. If there is no tracking failure the detector keeps adjusting the bounding box to the person's height and width.

(a) The quadcopter following a target to the right.

(b) The quadcopter getting closer to the target.

Figure 4.7: Detection (green rectangles) and tracking (red rectangles) in two different frames of the moving quadcopter.

Figure 4.8 is a five-frame figure showing how the quadcopter behaves when another person enters in the video. In the first three frames a person passes between the target and the quadcopter, and the tracker behaves correctly. But after that, the tracker detects the other person as the target and starts to follow it, as shown in the fourth and fifth frames. This could also have happened when the quadcopter was not moving, because the tracker, although working pretty good, is not perfect and can begin to track another person, although sometimes it can follow another person for a few seconds and then come back to the original target.

Summing up, on one hand, the tracker works pretty well and is able to follow the person even when the quadcopter is moving, but it can sometimes track another person that appears in the image. On the other hand, the detector is capable of readjusting the person's bounding box, and of detecting the person when a tracking failure occurs, although in this last case it may detect another person because the system is designed to search for a person in the whole image.

Figure 4.8: Consecutive frames showing a tracking failure recovery with a moving quadcopter and a change of the target.

4.2 Kalman filter evaluation

In this section we will test the implemented Kalman filter, comparing the observations given by the tracker and the corrections made with the filter. For these tests we will use the (x, y) pixel of the center of the bounding box of the tracker, and the filtered (fx, fy) pixel of the Kalman filter. All tests are created with the same initial parameters of the Kalman filter, a change in these parameters would produce a change in the behaviour of the filter. The used parameters are:

- $\mathbf{ax} = \mathbf{ay} = 5$
- $\mathbf{rx} = \mathbf{ry} = 10$
- $\mathbf{px0} = \mathbf{py0} = \mathbf{pu0} = \mathbf{pv0} = 20$

Figures 4.9 and 4.10 show the trajectories, through time, of the x coordinates and the y coordinates of the target, respectively, of both tracker and filter, with a landed quadcopter and a still target. We observe that, even when the target is still, the tracker creates some noise that is reduced with the Kalman filter.
In both figures, the horizontal axis is the time in seconds. In figure 4.9 the vertical axis shows the x coordinates of the target, and its range goes from the more to the left pixel, 167, to the more to the right pixel, 186. In figure 4.10 the vertical axis shows the y coordinates of the target, and its range goes from the bottom pixel, 57, to the top pixel, 64.

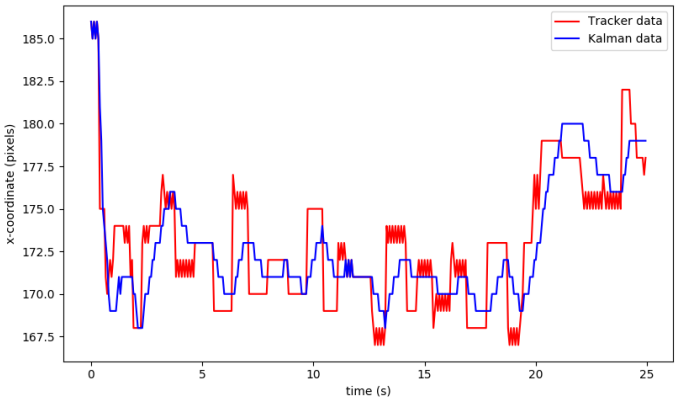


Figure 4.9: Filter and tracker trajectories of the x-coordinate through time with still target and quadcopter.

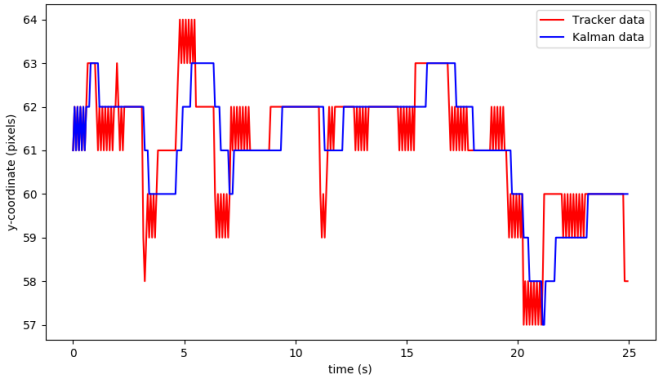


Figure 4.10: Filter and tracker trajectories of the y-coordinate through time with still target and quadcopter.

Figures 4.11 and 4.12 show, as the two previous figures, the trajectories, through time, of the x coordinates and the y coordinates of the target, respectively, of both tracker and filter, but this time the quadcopter and the target are moving (mostly back and forth). The filter reduces, in both figures, most of the noise created by the tracker.

The video tracking starts when the quadcopter is taking off, which makes the first increase in the y coordinate of the target. When the quadcopter stabilizes (at the fifth second approximately) the range in the y coordinate (from 85 to 110) is bigger than with the quadcopter stopped, but not in a too significant way. We can say the same about the x coordinate (range of which goes from 150 to 177 in the first 20 seconds), considering that the quadcopter is always receiving input to move left or right, the achieved stability is pretty good.

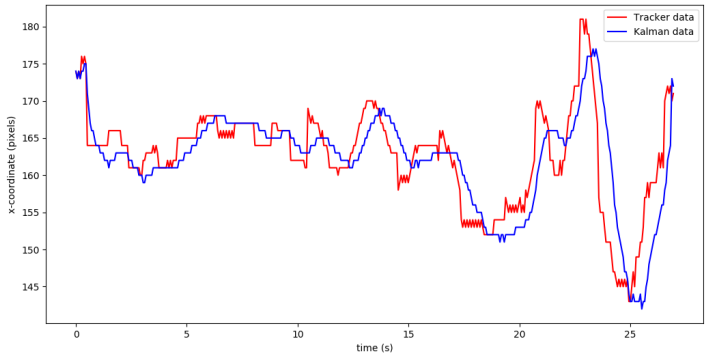


Figure 4.11: Filter and tracker trajectories of the x-coordinate through time with a moving target and quadcopter.

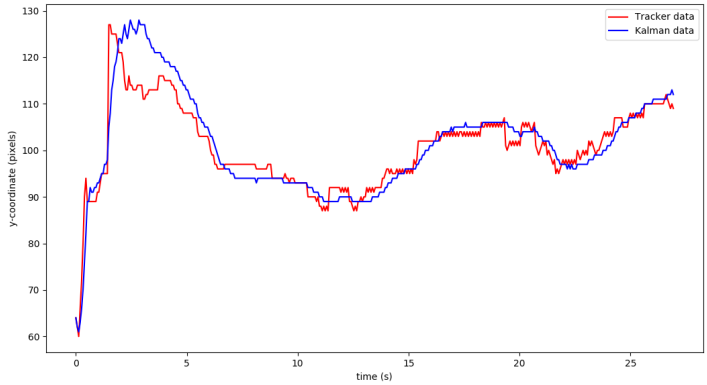


Figure 4.12: Filter and tracker trajectories of the y-coordinate through time with a moving target and quadcopter.

Finally, figures 4.13 and 4.14 show the correction obtained with the Kalman filter in a 85 seconds flight. In the first figure we can see how the person is moving from left to right several times. The filter, using our implemented model and the observation, creates a smoother movement when the person changes orientation. In the second figure we observe that the reduced noise is considerable.

In conclusion, the Kalman filter helps to create a smoother trajectory, by reducing the noise of the observations. This filtered observations are later on sent to the quadcopter and the smoother they are the better the quadcopter will move and behave.

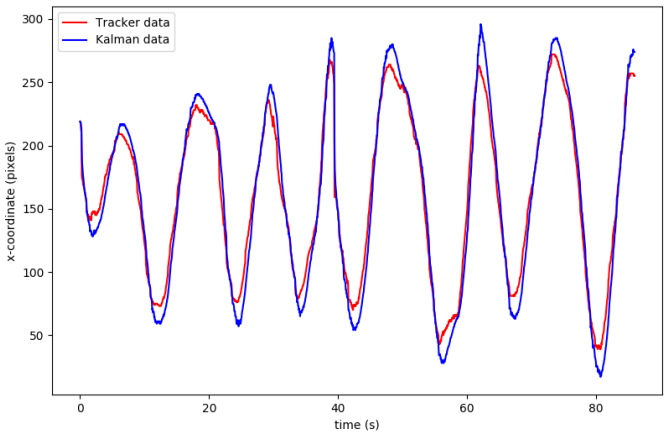


Figure 4.13: Filter and tracker trajectories of the x-coordinate through time with a moving target and quadcopter.

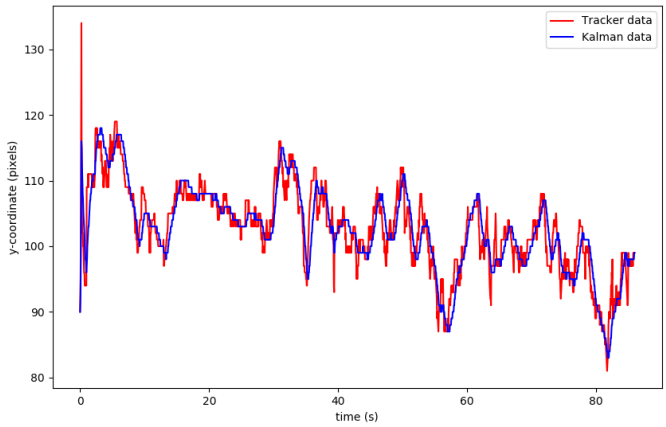


Figure 4.14: Filter and tracker trajectories of the y-coordinate through time with a moving target and quadcopter.

4.3 Quadcopter control evaluation

Up to now, we have tested the tracking algorithm and the Kalman filter and observed that both behave in a good enough manner for our purposes. In this last section we will evaluate the whole system, testing the final control of the quadcopter when tracking. As explained in section 2.3, each PID controller has to be tuned in order to suit the dynamics of the process. The quadcopter has four movements to be controlled: roll, pitch, yaw and throttle. In this project we have only implemented three of them: roll, pitch and throttle. For each of these movements a tuned PID controller has been created by means of the trial and error method. The first tests were made on the outside, but unfortunately the effects of the wind made impossible the tuning, because the AR.Drone 2.0 is not stable enough and the quadcopter kept moving from one side to the other. So the final tests, as observed

in the above images, were made indoors.

Before tuning each PID, a maximum speed was fixed to prevent the drone from going too fast. The final equation of the controllers is:

$$pid_k = vmax\ tanh\left(K_p e_k + K_i \sum_k e_k + K_d(e_k - e_{k-1})\right) \tag{4.1}$$

with $vmax = 0.4$. The parameters tuned for each of the controllers are:

- Roll
  - **Kp** $= 0.002$
  - **Ki** $= 0.$
  - **Kd** $= 0.04$
- Pitch
  - **Kp** $= 0.005$
  - **Ki** $= 0.00001$
  - **Kd** $= 0.05$
- Throttle
  - **Kp** $= 0.004$
  - **Ki** $= 0.$
  - **Kd** $= 0.$

Observing the tuned parameters we can say that for the roll movement we use a PD (Proportional-Derivative) controller, for the pitch movement we use a full PID control, and for the throttle movement we only use the Proportional part of the controller.

The drone movements are slow and unable to follow a person that is running or that rapidly walks out of the image, the problem can be in the not perfect adjustment of the parameters. Tuning the parameters is a very difficult task and depends a lot on the quadcopter. We could try to use these same parameters in another AR.Drone 2.0 and the movements may worsen. Moreover, the short time of the batteries (from 7 to 10 minutes) slowed down a lot the process.
However, a follow-me algorithm has been implemented and the tracking works pretty well: we have been able to create a software that detects a person in a room and sends to the quadcopter the needed commands to move and track, whenever the person is inside the visual field of the quadcopter's front camera, through the 3 dimensional space. The final movement of the quadcopter can be observed in the prepared video[1].

[1] https://youtu.be/ydnCSK7LFvU

5 Conclusions

This chapter is divided in two different parts. First we analyse the work done and then we introduce some improvements that can be done following the framework implemented here.

5.1 Summary and complications

This thesis has created an onboard vision system autonomous tracker for an AR.Drone 2.0 using three PID controllers. Moreover, an implementation of the Kalman filter has been created to reduce noise from the tracking. The tracking algorithm, combination of the MobileNet-SSD object detector and the KCF object tracker, ensures a non-stop tracking of a person, although sometimes a change of target can be produced. The follow-me quadcopter presented is able to track a person through the x, y, z coordinates in an indoor space or outdoor space with very little wind.

This framework is not able to follow a person completely due to the absence of the yaw movement, that could not be implemented. Although an important approach has been made and with some more work a better tracker could be created. However, quadcopters can be dangerous and the outdoor tests are very difficult: the AR.Drone 2.0 is a cheap quadcopter with limited stability and robustness against moderate winds. Because of this, creating a controller for the AR.Drone able to safely track a person in an outdoor space requires a lot of effort, time, and money to replace drones.

5.2 Future work

The implemented framework of this project can be improved and extended. Limitations of time, space and money have not allowed a perfect fulfill of the initial objectives. We present several points that can be used to carry on this work:

- The implementation of a Kalman filter for the depth movement. In this project, a Kalman filter following a constant velocity model was implemented to reduce the noise from the x and y coordinates of the frame. We propose a similar implementation to reduce noise in the z coordinate, obtained using the height of the bounding box at each frame versus the height of the bounding box at the first detection.
- The implementation of the yaw movement. The controlled movements in this project are roll (left-right), pitch (front-backwards) and throttle (up-down), but not the yaw movement that allows the quadcopter to turn right or left. This addition would let the quadcopter move in a complete trajectory through the 3D space.

- An improved tuning of the PID controllers parameters. The current controllers were tuned by moving the quadcopters in a *safe* way, making sure the velocities were not too high so someone could get hurt or the drone could break up. A better tuning can be made with more time, a bigger indoor space and the use of threads as ground anchors to constrain the quadcopter movement.
- The use of the CMT tracker instead of the KCF. The KCF tracker is used even though it is not the best tracker in the market. CMT is faster, more accurate, and more consistent to occlusions. But the only available python implementation of CMT uses OpenCV v2, while this project is developed in OpenCV v3.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL: `http://arxiv.org/abs/1603.04467`, `arXiv:1603.04467`.

[Bal12] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, and Daniel Silver, editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, volume 27 of *Proceedings of Machine Learning Research*, pages 37–49, Bellevue, Washington, USA, 02 Jul 2012. PMLR. URL: `http://proceedings.mlr.press/v27/baldi12a.html`.

[BYB09] B. Babenko, M. H. Yang, and S. Belongie. Visual tracking with online multiple instance learning. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 983–990, June 2009. `doi:10.1109/CVPR.2009.5206737`.

[Com16a] Wikimedia Commons. File:max pooling.png — wikimedia commons, the free media repository, 2016. [Online; accessed 27-May-2018]. URL: `https://commons.wikimedia.org/w/index.php?title=File:Max_pooling.png&oldid=204200335`.

[Com16b] Wikimedia Commons. File:pid en updated feedback.svg — wikimedia commons, the free media repository, 2016. [Online; accessed 8-June-2018]. URL: `https://commons.wikimedia.org/w/index.php?title=File:PID_en_updated_feedback.svg&oldid=220826562`.

[Com17] Wikimedia Commons. File:hyperbolic tangent.svg — wikimedia commons, the free media repository, 2017. [Online; accessed 18-June-2018]. URL: `https://commons.wikimedia.org/w/index.php?title=File:Hyperbolic_Tangent.svg&oldid=229436419`.

[Des16] A. Deshpande. A beginner's guide to understanding convolutional neural networks, 2016. [Online; accessed 18-May-2018]. URL: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/.

[DLHS16] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016. URL: http://arxiv.org/abs/1605.06409, arXiv:1605.06409.

[DT05] 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)*, volume 1, pages 886–893 vol. 1, June 2005. doi:10.1109/CVPR.2005.177.

[EREHJW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. 323:533–536, 10 1986.

[FS97] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997. URL: http://www.sciencedirect.com/science/article/pii/S002200009791504X, doi:https://doi.org/10.1006/jcss.1997.1504.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GDDM13] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL: http://arxiv.org/abs/1311.2524, arXiv:1311.2524.

[GGB06] H Grabner, M Grabner, and Horst Bischof. Real-time tracking via on-line boosting. pages 6–11, 01 2006.

[Gir15] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL: http://arxiv.org/abs/1504.08083, arXiv:1504.08083.

[HBS17] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. *CoRR*, abs/1705.02950, 2017. URL: http://arxiv.org/abs/1705.02950, arXiv:1705.02950.

[HCMB14] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-speed tracking with kernelized correlation filters. *CoRR*, abs/1404.7584, 2014. URL: http://arxiv.org/abs/1404.7584, arXiv:1404.7584.

[HDO+98] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, July 1998. doi:10.1109/5254.708428.

[HRS+16a] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016. URL: http://arxiv.org/abs/1611.10012, arXiv:1611.10012.

[HRS+16b] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016. URL: http://arxiv.org/abs/1611.10012, arXiv:1611.10012.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL: http://arxiv.org/abs/1704.04861, arXiv:1704.04861.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL: http://arxiv.org/abs/1502.03167, arXiv:1502.03167.

[Jac01] Paul Jaccard. Etude de la distribution florale dans une portion des alpes et du jura. 37:547–579, 01 1901.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014. URL: http://arxiv.org/abs/1408.5093, arXiv:1408.5093.

[KMM12] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(7):1409–1422, July 2012. URL: http://dx.doi.org/10.1109/TPAMI.2011.239, doi:10.1109/TPAMI.2011.239.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[LAE+15] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015. URL: http://arxiv.org/abs/1512.02325, arXiv:1512.02325.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. doi:10.1109/5.726791.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL: http://yann.lecun.com/exdb/mnist/ [cited 2016-01-14 14:24:11].

[NG06] A. Neubeck and L. Van Gool. Efficient non-maximum suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 850–855, 2006. doi:10.1109/ICPR.2006.479.

[NP15] Georg Nebehay and Roman Pflugfelder. Clustering of Static-Adaptive correspondences for deformable object tracking. In *Computer Vision and Pattern Recognition*, pages 2784–2791. IEEE, June 2015.

[PBED12] Stephane Piskorski, Nicolas Brulez, Pierre Eline, and Frederic D'Haeyer. *AR.Drone Developer Guide*. Parrot S.A., December 2012.

[RDGF15] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL: http://arxiv.org/abs/1506.02640, arXiv:1506.02640.

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL: http://arxiv.org/abs/1506.01497, arXiv:1506.01497.

[SEZ+13] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013. URL: http://arxiv.org/abs/1312.6229, arXiv:1312.6229.

[SIV16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. URL: http://arxiv.org/abs/1602.07261, arXiv:1602.07261.

[UvdSGS13] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013. URL: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013.

[VJ01] Paul Viola and Michael Jones. Robust real-time object detection. 57, 01 2001.

[WLF16] Min Wang, Baoyuan Liu, and Hassan Foroosh. Factorized convolutional neural networks. 08 2016.

[ZN93] J.G. Ziegler and N.B. Nichols. Optimum setting for automatic controllers. pages 759–768, 06 1993.