Università degli Studi di Milano-Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea Magistrale in Informatica

USI Università della Svizzera Italiana

Faculty of Informatics

Master in Informatics

Defining a New Controller for a Drone With User Partial Pose Estimation

Supervisor: Prof. Dr. Luca Maria Gambardella

Co-Supervisor: Dr. Alessandro Giusti

Co-Supervisor: Dr. Jerome Guzzi

Master Thesis of:

Dario Mantegazza

Student Number 736531

Academic Year 2017-2018

Contents

List of Figures

List of Tables

Listings

abstract

We propose a technique based on machine learning for controlling a drone equipped with a forward-looking camera, flying in proximity of an user. Using only the camera feed as input, the controller keeps the user in the center of the camera frame, and follows them as they move freely. We acquire training datasets specifically for this purpose, by means of an existing system which achieves the same goal but relies on a fixed infrared tracker that returns the pose of the user's head and of the drone. Using this system, we recorded more than 50 minutes of data with 13 different users, which resulted in 79739 training instances; each training instance consists in a camera image and the corresponding head pose relative to the drone. We then train a Convolutional Neural Network to estimate the latter from the former. In operation, our system applies the trained model to the current camera image, which yields an estimate for the user's head pose; this pose is fed to a controller that moves the drone accordingly. The system is validated on qualitative and quantitative real-robot experiments and simulations.

Introduction

This thesis describes the research and development processes that lead to the creation of an innovative system of Human-Robot interaction using a Machine Learning approach to solve a localization problem. The innovation is in the approach used in the problem solution.

Starting from an already implemented drone control system for indoor tracked flight and a demo that flies a drone making it always face a user, first we record some flight sessions and use them as data for training. We train and test a Convolutional Neural Network model to imitate the control system. In the past, Convolutional Neural Network have been successfully applied to drone control tasks [1] [2].

This project was developed at Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA) laboratories in Manno, Switzerland for the Master Thesis inside a double degree program between Universities of Milano Bicocca and USI, Università della Svizzera Italiana of Lugano.

The project was developed and researched from ground up, from initial data collection to a final real-drone test. The project was developed, trained and tested on a laptop with a GPU NVIDIA GTX 1070, perfect for Convolutional Neural Network training.

Document structure

The document is composed of six chapters and two appendices. Follows a brief description of each chapter.

Chapter 1 In this chapter we introduce the problem requirements and give a theoretical introduction to Machine Learning and Robotics arguments needed to understand our solution.

Chapter 2 In this chapter we describe the software tools that we use as scaffolding to build our system, the hardware utilized and the demo which we use in Section 3.2 to gather training samples using localization information from the tracker.

Chapter 3 To design our solution, first we analyze the data coming from the sources available to select which ones are useful as input and target for the CNN model that we use to predict the user's pose. With the selected sources we record raw data using the demo described in Chapter 2. Using video analysis we select the useful part of the recordings and generate two datasets, one used for fast iteration during the development of the model and one used for the evaluation described in

Chapter 5. With the newly generated dataset we train the CNN model and test it using 5-fold cross-validation.

Chapter 4 The system developed is composed by multiple scripts handling different functionalities. Different libraries and framework are used in the code or by the system. In this chapter we briefly introduce libraries and frameworks and discuss the implementation of the system.

Chapter 5 In this chapter we discuss the evaluation of our solution.

Chapter 6 Here we discuss the conclusion and future work.

Appendix A Listings of our implementation source code.

Appendix B Additional Figures.

Chapter 1

Problem specification

In this chapter we introduce the problem requirements and give a theoretical introduction to Machine Learning and Robotics arguments needed to understand our solution.

1.1 Requirements

At IDSIA, in Manno, Switzerland, researchers have developed an interesting demo [3] of Human-Robot interaction. The demo, left image in Figure 1.1, works using an external fixed Infrared (IR) tracker that returns absolute poses of drone and user. The demo's controller computes the relative pose of the user with respect to the drone and uses it to make the drone fly in front of the user at a distance of 1.5 meters. This demo needs a room with a tracking hardware that is installed on the room's walls to accurately localize drone and user. This is a limitation; in a more realistic scenario this tracking system wouldn't be available and such demo would not work.

The advantages of having a drone facing the user are many. For example a drone flying at eye's height is more visible than a drone flying many meters above the user. A drone could wait user's commands flying facing them; this could reduce fear of drones in users.

The objective of this thesis is to reproduce the same demo using the drone's front facing camera to predict the user's relative pose instead of the external tracker. To reach this objective, we collect multiple images associating them to the user relative pose computed from the external tracker's data. We use this data to train a Convolutional Neural Network model. After training, the Convolutional Neural Network predicts in real time the relative pose of the user for each image captured by the drone's frontal camera; right image in Figure 1.1 illustrates our demo.

1.2 Machine Learning introduction

Machine Learning, a branch of artificial intelligence, is a category of algorithms that uses statistical techniques and computational methods to give software applications the ability to learn directly from data. A computer that is able to learn means that it can improve performance and accuracy on a specific task directly from the data without being explicitly programmed or relying on a predetermined equation as a model. Machine

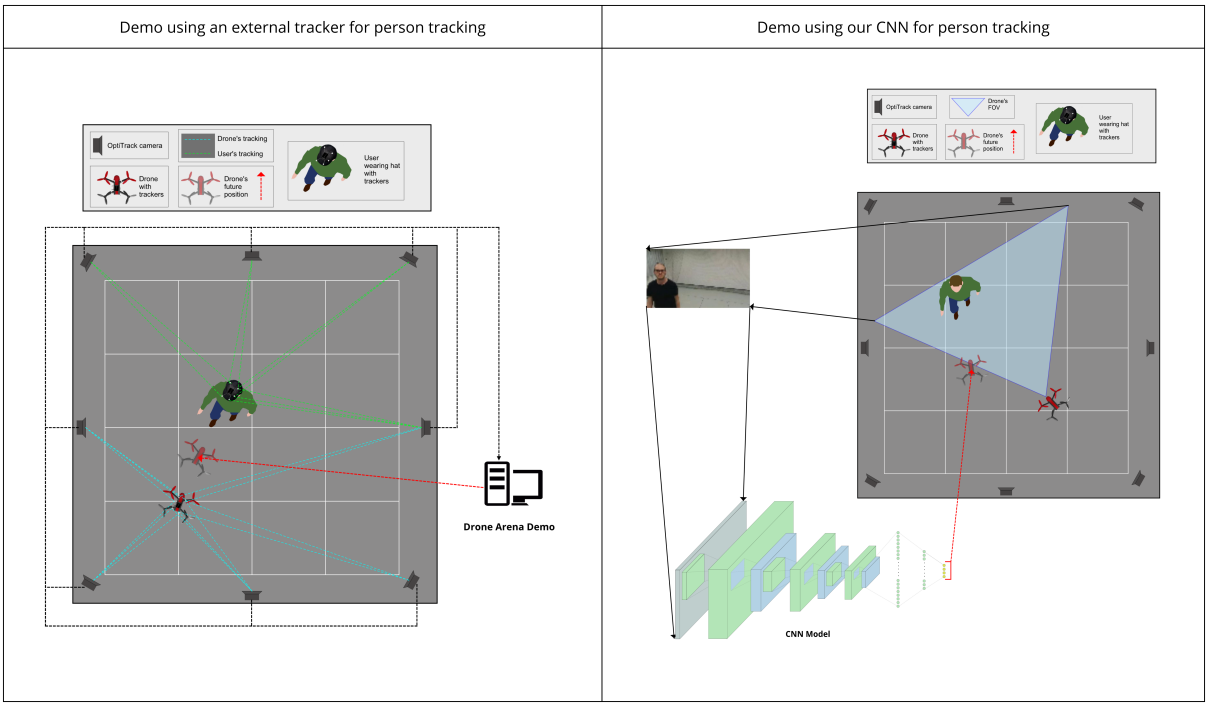


Figure 1.1: The Drone Arena demo and our solution compared.

Learning tasks can be categorized as supervised or unsupervised, the main difference between the two is the presence of a feedback on the predictions.

Supervised Supervised learning [4] is the category in which fall all the techniques that present to the model both a known input and desired output with the goal of learning to generate reasonable predictions for a given input. During the training phase, an "Expert" provides feedback to the model improving the model's ability to correctly predict.

Supervised learning is used whenever there is knowledge of both input and output. If the feedback signal is only in form of rewards or punishment, then we talk about Reinforcement Learning.

Unsupervised Learning Unsupervised Learning [4] algorithms don't rely on a feedback signal, the model has to find on its own patterns or intrinsic structures in the input.

The solution used for our problem falls under Supervised Learning.

Another categorization of Machine Learning algorithms is on applications. The most typical applications of Machine Learning are classification and regression.

Classification Classification [4] techniques predict a discrete output. Classification divides the inputs in two or more classes. The model is trained to predict those classes even for unseen inputs. A typical example of a classification problem is recognizing if in an image is present a dog or a cat. Another example of classification is spam filtering; emails must be classified either as spam or non-spam. In case the classes are not known in advance then we talk about Clustering.

Regression Regression [4] techniques predict a continuous output. These techniques are used if the nature of the system is continuous, a real number. Typical examples of regression problems are stock prediction or power grid load.

Given the nature of our problem, the solution falls under regression category.

Machine Learning is a category of algorithms, depending on which type of problem we have to tackle and what type of solution we want to develop, different algorithmic approaches are available. If we are performing a classification task, then common algorithms are: Support Vector Machines (SVM), decision trees, k-nearest neighbor and Neural Network (NN). Instead, if we want to solve a regression problem, common methods include: linear and non linear models, Bayesian linear regressors and Neural Network. For both, Neural Network are a possible solution and in recent years, Neural Network and derived techniques as CNN, Recurrent Neural Network (RNN) [4] and Deep learning are becoming more and more the chosen solution. Deep Learning indicates every kind of NN architecture with more than one hidden layer, even if a formal definition for the concept doesn't exist.

1.2.1 Artificial Neural Networks

Even if Artificial Neural Networks are not new [4], only recently they had an increment in usage, expanding into new and untested areas of application.

Neural Network [4] are vaguely inspired by biological neural networks because they have been motivated right from the beginning by the recognition that human brain computes in a completely different way from common computer. The brain is a complex, nonlinear and parallel computer with the capability to organize its constituents: the neurons. The brain has the plasticity to adapt its nervous system to it's environment.

In general, a Neural Network is a complex, nonlinear, adaptable and parallel algorithm that employs a massive interconnection of simple computing cells often called "neurons". Similarly to a brain, a Neural Network is a machine that is able to adapt itself to the task provided by changing its own inter-neuron connections. These connections have a linked strength called synaptic weights, or simply "weights". By changing the values of these weights, a Neural Network is able to store knowledge about its environment, a simple NN is visible in Figure 1.2. In recent years multiple architectures of Neural Network have been developed in relation to specific goals, but all NN share some core concepts.

Artificial neurons The core component of an artificial neural network is the artificial neuron [4]. Usually a NN composed by a single artificial neuron is called Perceptron. Inspired by real neurons, artificial neurons are a computational node that resemble a non linear function. Artificial neurons receive as input a vector of features, multiply every respective element of the input vector with a weight and the results are summed. Usually an extra input value set as 1 is associated to a *special* weight called bias, the bias is used to influence neuron learning. The sum result is passed to an Activation Function and the

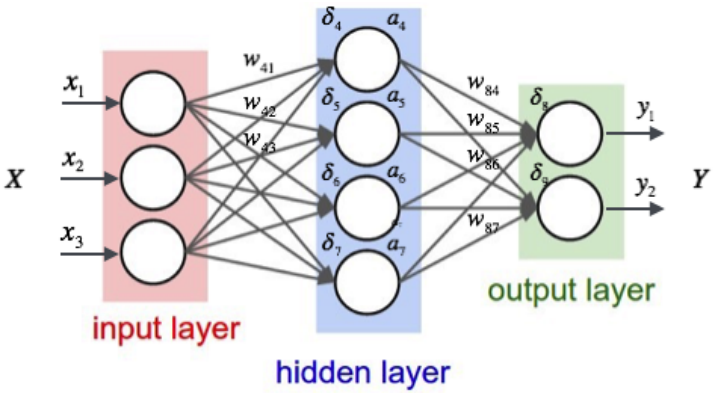


Figure 1.2: A schematic representation of a neural network with one hidden layer.

function's result is the final output. This process can be expressed with an equation:

$$y = \theta(z) \tag{1.1}$$

$$z = \sum_{0}^{n}(x_i \cdot w_i) + b \tag{1.2}$$

y is the final prediction value, θ is the activation function, b is the bias, x_i is an element of the input vector X and w_i is an element of the weights vector W. A schema of a Perceptron is visible in Figure 1.3. Perceptrons optimize their parameters, the weights and bias, through an iterative optimization process composed of two steps:

Forward pass in this step an input vector is fed to the Perceptron and the output is computed,

Backpropagation pass the predicted output is compared with the target label generating an error, which is used to minimize a loss function, updating the Perceptron's weights accordingly. The optimization process is responsible for the weights change and is carried out by the Optimizer algorithm.

Activation functions The activation function is a core component for Perceptrons and layers of a NN. The activation function is used to introduce non-linearity into the system. The non-linearity is needed by the NN to be able to produce non-linear predictions; many real life problems are non-linear. A non-linear problem is a problem that cannot be solved with a linear combination of inputs, example the XOR problem [4].

The most common activation functions are:

- Sigmoid function

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-}x} \quad , \sigma : \mathbb{R} \to (0, 1) \tag{1.3}$$

- Hyperbolic tangent function

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad , \ \tanh : \mathbb{R} \to (-1, 1) \tag{1.4}$$

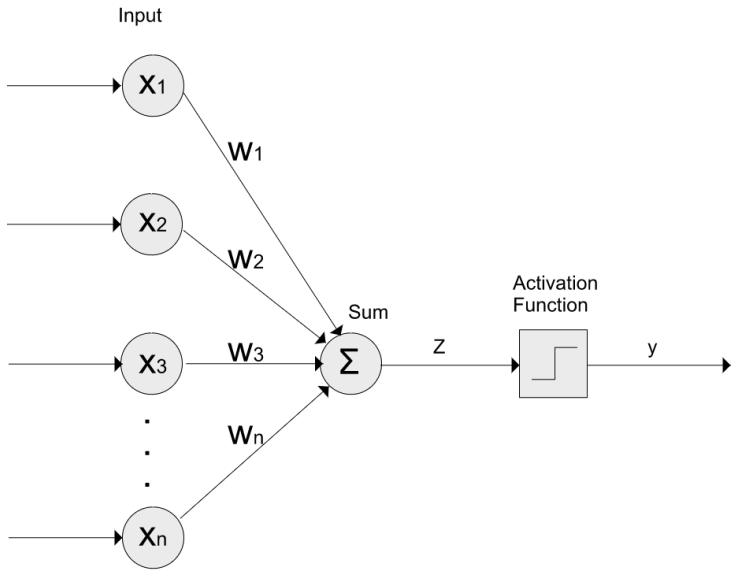


Figure 1.3: A representation of Perceptron.

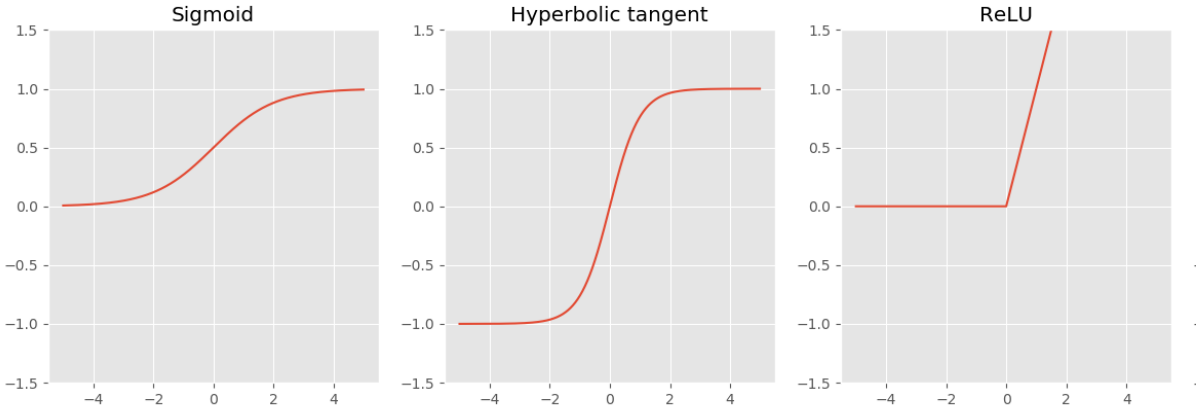


Figure 1.4: Three plots of the three different activation functions.

- Rectified Linear Unit (ReLU) function

$$f(x) = \max(0, x) = \begin{cases} 0, & \text{for} \quad x < 0 \\ x, & \text{for} \quad x \geq 0 \end{cases} \quad , f : \mathbb{R} \to [0, \infty) \tag{1.5}$$

- Identity function

$$f(x) = x \quad , f : \mathbb{R} \to \mathbb{R} \tag{1.6}$$

The first three functions are visible in Figure 1.4

Optimizers and loss function An optimizer is an algorithm that tries to minimize or maximize its objective function. For the NN the objective function is the loss function and the optimizers used try to minimize it. One of the first optimizer used for NN is the Gradient Descent [4].

A loss function [4] is a function that maps one or more values to a real number corresponding to a *cost* or *error*. Usually the loss function is MSE for classification [4] problems and MAE for regression, but many other are available. In our project we utilize Adam optimizer with Mean Absolute Error (MAE) as loss function, both are described later in Section 3.4.2.

Layers A layer is a set of artificial neurons using the same activation function. Neurons in a layer are connected only to other neurons of a different layer. Usually layers are categorized as:

Input Layer A layer that receives as input the data as multi-dimensional vector.

Hidden Layer Any layer that is not an input or an output layer, its input is data coming from previous layer and its output is fed to successive layer.

Output Layer A layer whose output is the prediction of the NN.

More complex architecture of NN still use the same technique for learning their parameters in order to solve their problems. Our solution utilizes a Convolutional Neural Network model and even if it has a different and more complex architecture compared to a single hidden layer NN, what discussed until now is compatible.

1.2.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) [4] have reached very high performance in computer vision and pattern recognition [5] and are now the standard [4] in these fields. CNN have been already used for drone control problems [1] [2].

CNNs are another example of Neural Networks inspired from nature. From Hubel and Wiesel's early work on the cat's visual cortex [6], we know that the visual cortex contains a complex arrangement of cells sensitive to small sub-regions of the visual field (receptive fields). Tiled to cover the entire visual field, these cells act as local filters over the input space reacting to strong spatial local correlation present in natural images.

CNNs are a parametrized functions; similar to any other Machine Learning technique, these parameters can be optimized to minimize a cost function using gradient descent.
Convolutional Neural Networks are usually composed of three type of layer: convolutional layers, (max)pooling layers and fully connected layers, a schematic representation of a CNN is visible in Figure 1.5.
CNN differs from a feed forward Neural Network on how neurons are connected between layers. In a convolutional layer neurons are not necessarily connected to all the other neurons from the previous layer but only to neurons in a particular window. A neuron in a convolutional layer is replicated through weights sharing. Artificial neurons in a convolutional layer are grouped in *filters* and they react to specific spatial local patterns, similarly to the receptive fields in visual cortex.

1.3 Robotics Introduction

In this section we introduce the very central concept of robot pose.

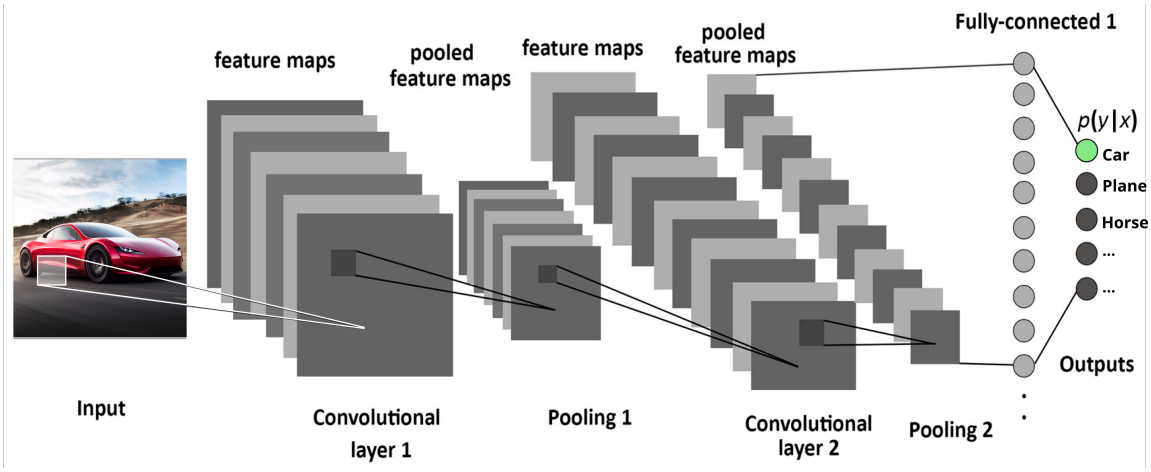


Figure 1.5: A representation of CNN solving a classification problem.

1.3.1 Pose

The drone and user location returned by the external optical tracker is parametrized by a 6D pose. The pose [7] is a concept representing the position and orientation in 3D space of an object or a person with respect to (wrt.) a reference frame. In our case the reference frames are the room, in which we test our system, and the drone.

A pose can be represented in different ways but three core concepts are always present:

- the 3D position, a triple composed of the three values corresponding to the a point projection on the three axis, as in Figure 1.6;
- the 3D orientation;
- a frame of reference.

The 3D rotation can be represented either with a quaternion or angles.

In our project we use the quaternion and Euler angles representation of the rotation, depending on the task at hand. If we represent the rotation with Euler angles and its relative 3D rotation matrix ${}^{\text{world}}R^{3\times3}_{\text{object}}$, then it is possible to define a 4x4 homogeneous roto-translation matrix:

$$ {}^{\text{world}}T_{\text{object}} = \begin{bmatrix} {}^{\text{world}}R^{3\times3}_{\text{object}} & t^{3\times1} \\ 0^{1\times3} & 1 \end{bmatrix} \tag{1.7} $$

where $t^{3\times1}$ is the translation column vector composed of the three 3D position components.
In order to change the frame of reference of and object A from *world* to the frame of reference of object B, we can use the pose composition, defined as follows:

$$ {}^{\text{B}}T_{\text{A}} = ({}^{\text{world}}T_{\text{B}})^{-1} \cdot {}^{\text{world}}\, T_{\text{A}} \tag{1.8} $$

where $\cdot$ is the matrix multiplication and T^{-1} is the inverse matrix.

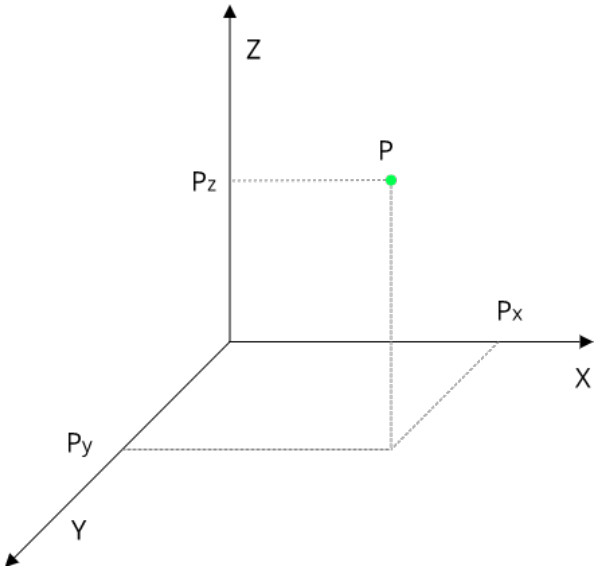


Figure 1.6: Point P projection on three axis. The position of point P are (P_x, P_y, P_z)

Chapter 2

System description

In this chapter we describe the software tools that we use as scaffolding to build our system, the hardware utilized and the demo which we use in Section 3.2 to gather training samples using localization information from the tracker. We will later feed, in Section 5.2, the person's relative pose predicted by our CNN to the same controller as an alternative source of localization.

2.1 Hardware

Here follows an in-depth description of the project's hardware.

2.1.1 Parrot Bebop 2

The drone used for this project is the Parrot Bebop 2, a lightweight drone[1] capable of indoor or outdoor flights. The Parrot Bebop 2 can achieve bursts of speed of 60 km/h horizontally and 20 km/h vertically. It reaches its maximum speed in 14 seconds, brakes in 4.5 seconds and resists head winds up to 63 km/h.

The drone is powered by a swappable battery with a capacity of $2700mAh$. A new, fully charged, battery can last for a 25 minutes maximum flight. An older, more used battery may only last up to 10 minutes of flight. An empty battery is easily swappable for a charged one in a matter of minutes[2] and multiple batteries are available.
Usually the Parrot Bebop 2 is controlled via smartphone or through the "SkyController", which is Parrot's two-stick remote control for the Parrot Bebop 2. The controller creates a Wireless Local Area Network (WLAN) using the protocol Wi-Fi 802.11-n dual-band, one of 2.4 GHz and one of 5 GHz. This network is provided by two high performance dual-band Multiple-Input and Multiple-Output (MIMO) antennas.

For the system used in the project the drone is connected to the Drone Arena computer directly through Wi-Fi.
Unfortunately the Parrot Bebop 2 suffers of connectivity problems [8] [9], these problems can result in video feed degradation, video feed loss or inability to control the drone, in section 3.3.1 a workaround for this problem is described.

[1] The drone weights only 500 grams
[2] Counting the shutdown and restart time of the drone after the battery swap

Figure 2.1: An image of the Parrot Bebop 2 "exploded" showing the internal components

Drone's sensors

The Parrot Bebop 2 is equipped with seven different sensors:

- A vertical stabilization camera takes an image of the ground every 16 milliseconds and compares it to the previous one to determine the speed of the drone.
- An ultrasound sensor analyzes the flight altitude up to 5 meters.
- A pressure sensor measures air pressure and analyzes flight altitude beyond 5 meters.
- 3-axis gyroscope measures the attitude of the drone.
- Accelerometer measures the positioning of the drone on 3-axis and its linear speed.
- 3-axis magnetometer helps define the orientation of the drone, like a compass.
- Global Navigation Satellite System (GNSS) chipset (Global Positioning system (GPS) + GLONASS) geo-localizes the drone and helps measure the speed in order to stabilize the drone at high altitudes. This sensor is not used in this project

Sensors data are collected and computed by the on-board computer[3] and used for flight control.

Front facing camera

The Parrot Bebop 2 is equipped with a 180° FOV 14 mega-pixel front facing camera capable of recording at 30fps, an image of the camera can be seen in Figure 2.2. This

[3]The drone controller has a quad-core CPU

Figure 2.2: An image of the Parrot Bebop 2 front facing camera

camera is used by the drone to generate a hemisphere dome that represents the raw camera feed. On the hemisphere surface, a virtual window is moved and rotated to produce the software-stabilized video feed. The resulting video feed is a full-HD 16:9 ratio image with a horizontal FOV of 90°.
With this technique the on board controller compensates drone's pitch and roll movement trying to get always horizontal images.

2.1.2 OptiTrack

A Motion Capture (MoCap) system is a system that is able to precisely measure movements of objects or people. MoCaps are used in a variety of applications, from entertainment to medical research. There are different ways to implement a MoCap system but usually they all use multiple cameras pointing to the same space in a room from different points of view. These cameras, one shown in Figure 2.4, record data both as visible light information, a normal RGB image[4], and as IR image. The latter is used to track trackers present in the FOV of the camera.

The trackers can be either active or passive. The OptiTrack system available at IDSIA uses passive trackers; an example of a tracker is shown in Figure 2.5. OptiTrack is a state of the art MoCap system used by a wide range of companies and research labs. OptiTrack systems can deliver high performance 3D motion capture with ultra low latency real-time output, with free and open developer access. OptiTrack real time tracking systems are used for their low latency and precise 6 Degree of Freedom (DoF) tracking system for ground and aerial robotics.

In the Drone Arena, the OptiTrack setup is composed of 12 cameras, the virtual fence is a 4.8×4.8 meters area, while the tracked area is 6×6 meters. A similar setup, shown in Figure 2.6, can capture up to 18 drones in a Capture volume up to $7 \times 7 \times 2$ meters with a $100\mu m$ precision.

[4]Not used in this project

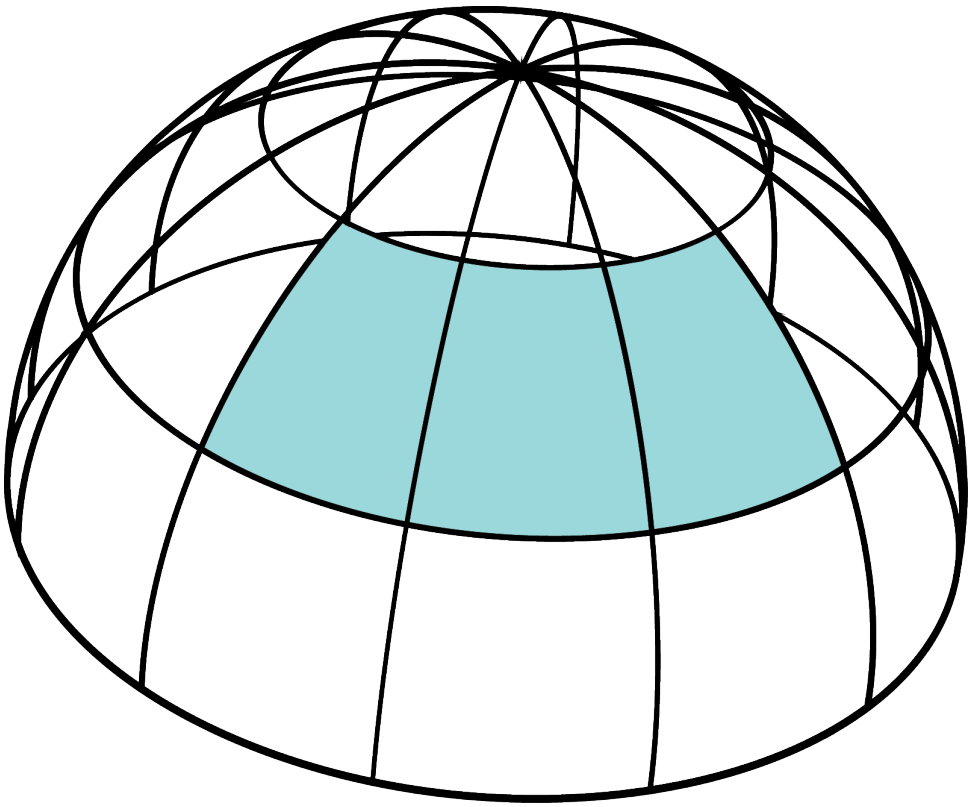


Figure 2.3: A representation of the digital stabilized view of the drone; the cyan area represents the actual camera frame and the wire-frame dome, the 180° FOV of the drone's camera.

Figure 2.4: The OptiTrack Prime 13 camera present in the Drone Arena.

Figure 2.5: The image illustrates a reflective passive tracker.

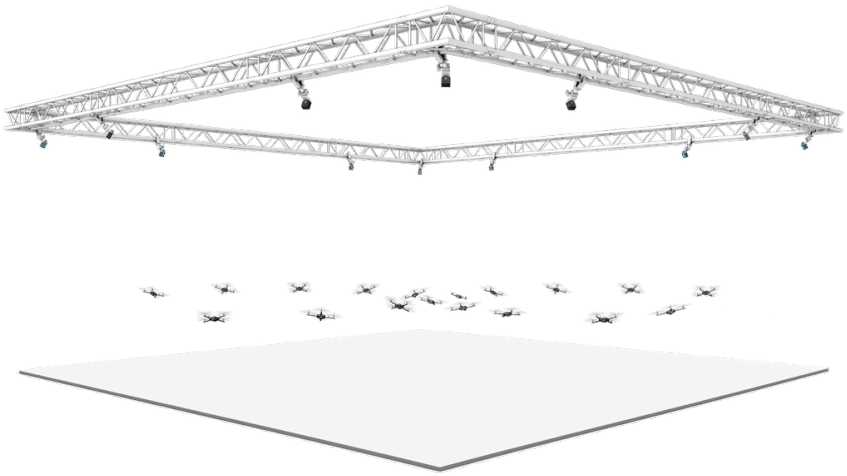


Figure 2.6: A setup of 12 Prime 13 OptiTrack cameras, similar to the one present in the Drone Arena. A similar setup is able to track up to 18 drones.

2.2 Software

In this section we discuss software tools we use to build our system. Our scripts and relative libraries are discussed in Chapter 4.

2.2.1 Robot Operating System (ROS)

ROS is a open-source[5] meta-operating system for robots [10]. ROS is the industry and research standard framework for robotics and it is aimed to help software developers to create robotic applications. Its primary goal is to support code reuse in robotics research and development. Even if ROS is not an actual Operating System (OS), it provides the services you would expect from an OS such as: hardware abstraction, device drivers, message-passing, package management and more.

Even if it is only officially supported on Ubuntu Linux, multiple community supported variants are available for other Unix OS and an experimental version is available for Microsoft Windows.

The main features of ROS are:

- a high-level hardware abstraction for sensors and actuators;
- an extensive set of standardized messages types and services;
- a peer-to-peer network architecture with connection broker;
- generation of a computational graph;
- fault-tolerance: computing units(nodes) are separated OS processes.

Computational graph

ROS computational graph is a peer-to-peer network of processes loosely coupled using the ROS communication infrastructure.

The graph is composed by:

- `roscore`
- Nodes

roscore `roscore` is a collection of nodes and programs that are pre-requisites. It is needed to have a `roscore` running in order for ROS nodes to communicate.
When started, `roscore` launches:

- ROS Master: a connection broker essential for the computational graph, it provides name registration and lookup for nodes, topics and services.
- ROS Parameter Server: acts as a central location for storing data by key.
- `rosout`: a logging node used as a broadcasting facility.

[5]Main client libraries and tools are under the terms of the BSD license (`https://en.wikipedia.org/wiki/BSD_licenses`).

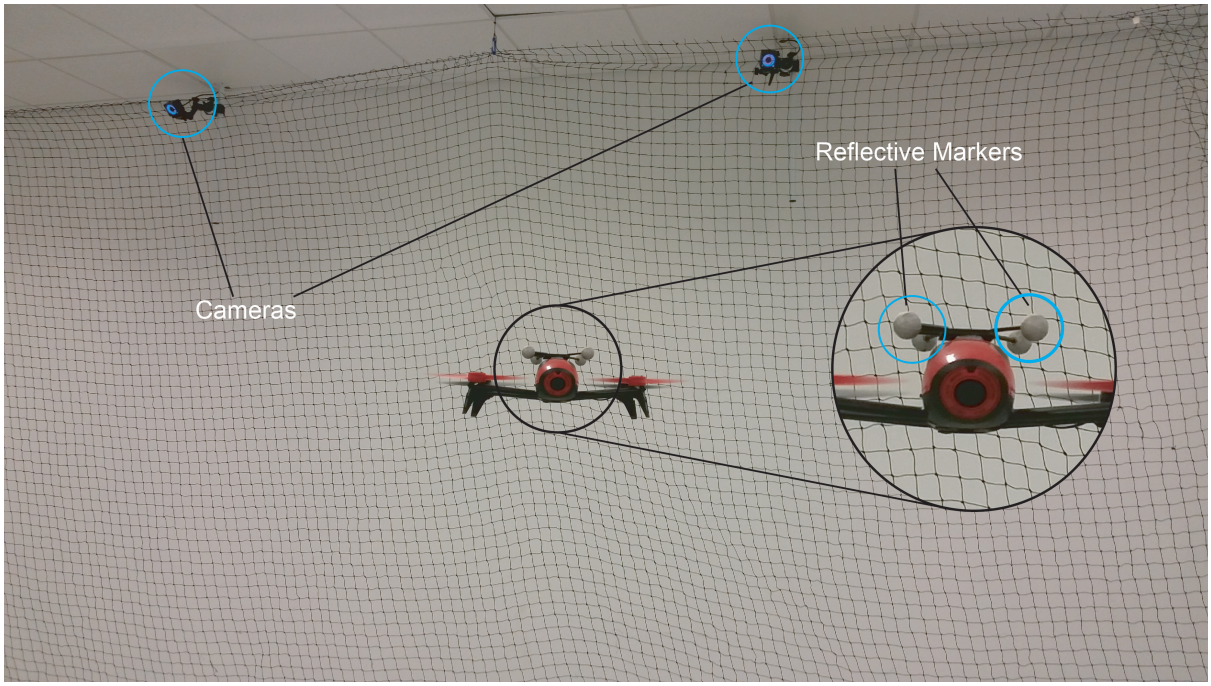


Figure 2.7: A view from the user point of view inside the drone arena. The drone reflective markers are visible.

Nodes ROS nodes are executable files that use ROS to communicate with other nodes. Nodes are usually fine grained processes that perform precise computations. To communicate, nodes use a ROS client library to publish or subscribe to *topics* or *services*. Different Nodes can run on different hardware while being in the same ROS system.

rosbag

Another function of ROS, used multiple times in this project, is `rosbag` files. These files are the primary mechanism for data collection in ROS.
Each rosbag file contains all the messages from different ROS topics recorded during a session.

2.2.2 Drone Arena

The pre-existing system used for collecting data is located in a dedicate room inside IDSIA. Inside the room there is an OptiTrack system composed of multiple motion tracking cameras and a drone.

The drone and the user carry reflective markers, Figure 2.7. Using these markers, the tracking cameras can define the poses of the user and the drone inside a virtual fence [3], the OptiTrack system is described in Section 2.1.2.

To fly the drone inside the Drone arena, there is an already developed software that controls the drone to make it always face the user face, as can be seen in figure 2.7, also the controller avoids that the drones flies over the virtual fence.

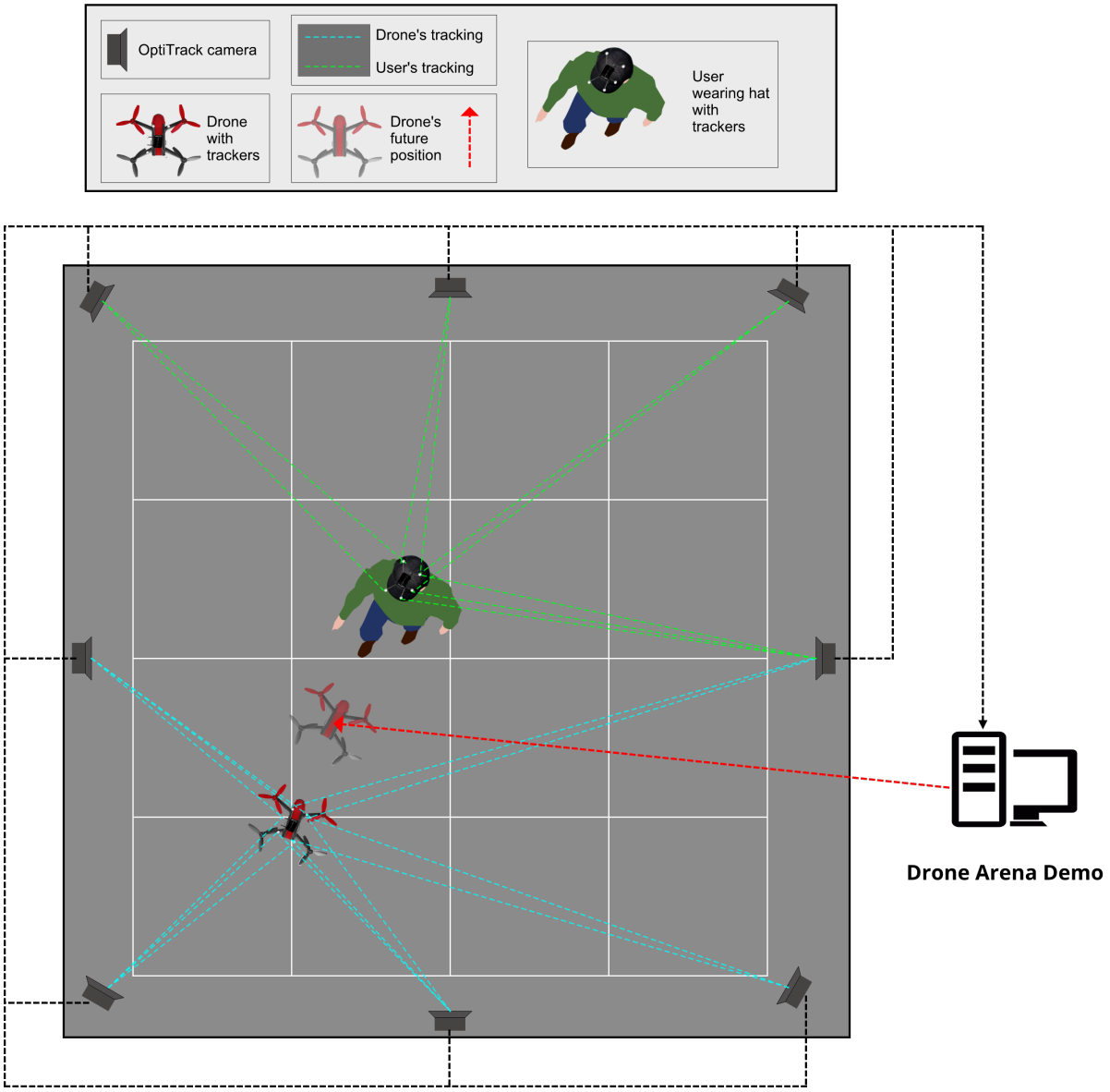


Figure 2.8: A schematic visualization of how the demo works.

2.3 Demo

The demo uses the Drone Arena controller to post to the drone the target pose computed by the Drone Arena using the drone and user data coming from the OptiTrack. The drone while its flying tries continuously to face frontally the user while staying at a predetermined distance and at a height difference. To do so, the demo uses the user's pose and drone's pose captured by the OptiTrack system, computes the drone's target pose inside a virtual fence and sends accordingly to the drone new accelerations commands. An illustration of how the demo works is in Figure 2.8.

Chapter 3

Solution design

To design our solution, first we analyze the data coming form the sources available to select which ones are useful as input and target for the CNN model that we use to predict the user's pose. With the selected sources we record raw data using the demo described in Chapter 2. Using video analysis we select the useful part of the recordings and generate two datasets, one used for fast iteration during the development of the model and one used for the evaluation described in Chapter 5. With the newly generated dataset we train the CNN model and test it using 5-fold cross-validation. Figure 3.1 illustrates a schematic representation of the dataset recording and creation.

3.1 Data analysis

3.1.1 Data sources

To be able to train a Machine Learning model and reach the proposed objective, an accurate selection of the data and data sources is needed.
The initial approach to the problem can be divided in the following steps:

- select the correct hardware;
- gather samples of data from selected hardware;
- represent and analyze the data recorded to decide which data sources to use.

This approach is followed by raw data recording and dataset creation.

Selecting the hardware

For this project the hardware selection is constrained by the requirements of the project and by the availability of hardware resources at IDSIA. The hardware selected is described in Section 2.1.2.

Gathering a sample of data

With this hardware it is possible to setup a simple recording session with a user moving around the flight-area with the drone always facing the user, while tracking user and drone with the OptiTrack. Every message present on every ROS `topic` is recorded in a `rosbag` file for later analysis.

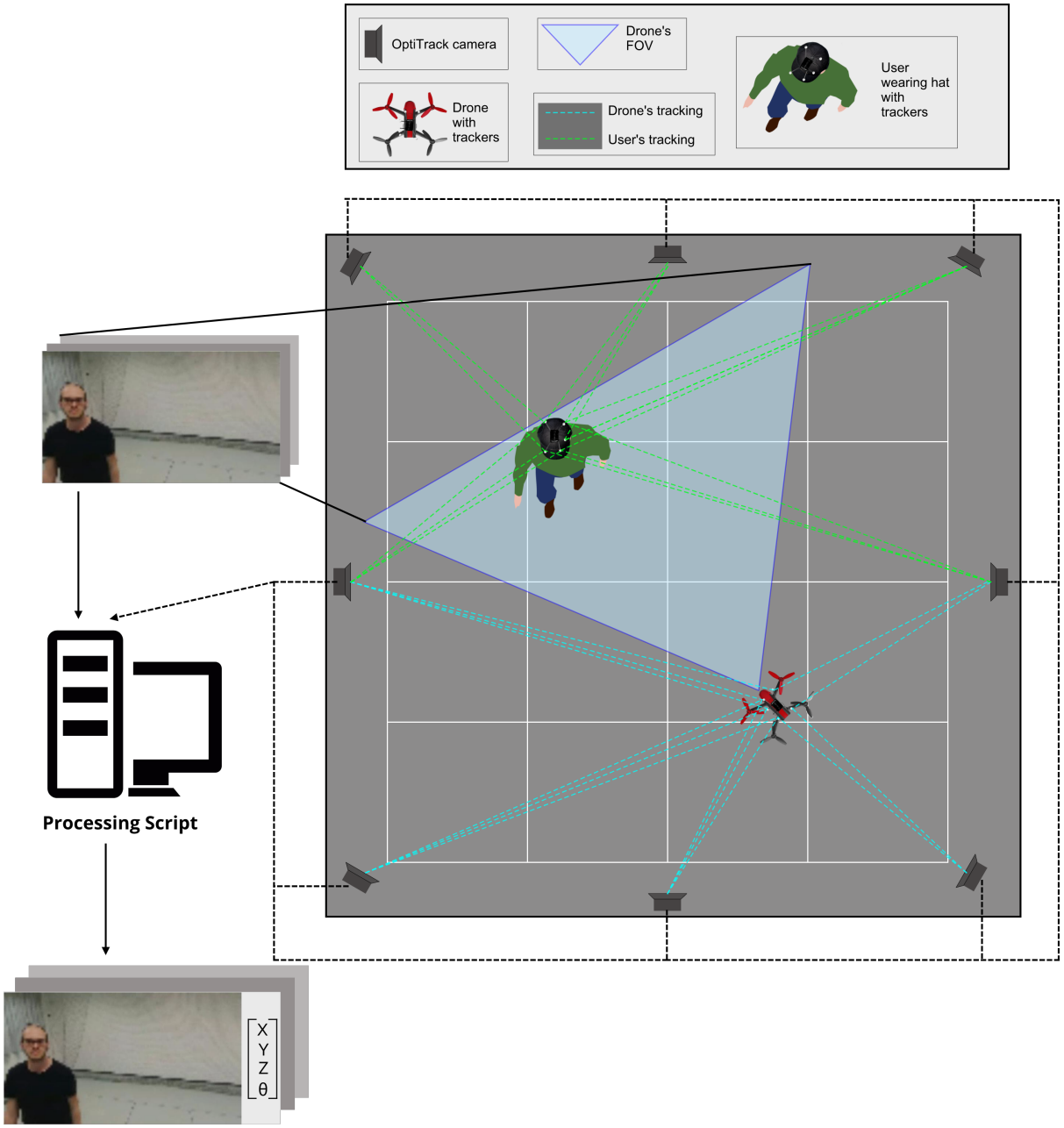


Figure 3.1: An illustration of how we collect and process the dataset.

Represent and Analyze the data recorded

After understanding what data is recorded, how it is recorded and how each ROS `topic`'s message can be extracted, a custom written Python script is used to create a representation of the different data sources: camera feed is presented as a video and odometry data is shown either with text or some type of plot.

An in-depth look at the data representation highlighted the following facts:

Video format The video feed is available in two different data formats, but the differences between the corresponding `topic`'s messages, `\bebop\camera_raw` and `\bebop\camera_raw\compressed`, are only related to how the video frames are saved and not to how the image is compressed. In fact the image data is the same but the `\bebop\camera_raw\compressed` saves the same data more efficiently, without losing information.

Wi-Fi disconnections As described in section 2.1.1, the Bebop communicates with the Drone Arena system through Wi-Fi, sometimes this signal deteriorates thus the video feed deteriorates. In the best case scenario the video deteriorates partially and is still usable, an example is shown in Figure 3.2; this data is still used because a similar situation is expected to happen in the evaluation phase. In the worst case scenario the video feed completely deteriorates and either the video becomes black or freezes on a frame; these frames are dropped.

OptiTrack calibration If the OptiTrack system present in the Drone Arena is not calibrated, sometimes the odometry feed is not updated even if the user or the drone are moving inside the tracked volume. Fortunately this type of error usually happens at the begin of a recording session, a simple solution to this problem is detailed later in section 3.3.3.

Synchronization The three `topics` that we are going to use are not synchronized. The computer that runs the OptiTrack system is not synchronized with the drone internal controller, the odometry and camera messages are out of sync. In addition, the camera `topic` and the odometry `topics` have different recording rates thus the data recordings need to be synchronized, a representation of this problem is shown in Figure 3.3, and a solution to this problem is detailed later in section 3.3.1.

Missing data Usually the video feed is the most sparse and less numerous of the three data stream. The sparsity of the data is not related to the frame-rate but it is caused by connection issues.

Data sources selection

After visualizing and analyzing the sample data, three `Topics` are chosen as data sources.

The three data sources selected are:

- Drone's odometry in the frame reference of the Drone Arena[1] as ROS messages on the *topic* `/optitrack/bebop`, the Drone pose data coming from the OptiTrack system.

[1]Interpreted for this project as *World* reference frame

Figure 3.2: Six examples of deteriorated video feed. This type of data is still used as part of the final dataset.

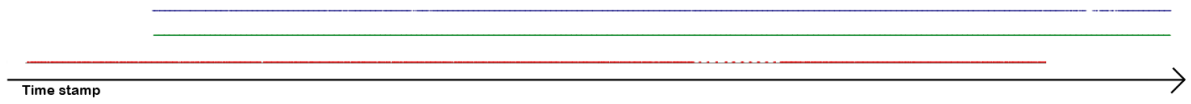


Figure 3.3: This figure represents an example of the messages distribution of three different topics. Blue and green are odometry topics and red is the camera feed topic. The lines are composed by dots, each dot is a message.

- User's head odometry in the same frame of reference as ROS messages on the *topic* `/optitrack/head`, the users head pose data coming from the OptiTrack system.
- Drone's front facing camera feed, described in Section 2.1.1 on the *topic* `/bebop/image_raw/compressed`, the JPEG compressed images coming from Parrot Bebop 2 front facing camera

These sources are selected because they are going to be used as input for the CNN (the drone's camera feed) and as raw data(the two poses) to compute the corresponding labels; a representation of the dataset creation process is visible in Figure 3.1.

Odometry data Through the MoCap system described in Section 2.1.2, it is possible to gather positional information about every marker in-sight applied to an object or person.

This information is used to compute the odometry data of any objected or person with a set of reflective markers. The odometry data is then published on ROS `topics` as a message of type `PoseStamped`, this message type is described in Section 4.3.1.

Camera data The camera information, and RGB matrix, coming from the drone is a ROS message of type `CompressedImage` .This message is briefly described in Section 4.3.1.

3.2 Data collection

After this first analysis the data and sources are selected and the raw data gathering is started. The final dataset required twenty two recording sessions.

In each recording session a MoCap hat, Figure 3.4, or headband fitted with reflective markers is worn by the user. Then the user is instructed to enter the Drone Arena and after a briefing on what behavior to expect from the drone inside the Drone Arena, we let the drone fly and starts to follow the user's head as described in the Section 2.3. In the meantime on a computer connected to the ROS computational graph, a `rosbag` is recorded saving only the selected topics. During a session, the user would walk inside the Drone arena while the drone tries to face them. In order to reduce possible biases in the data, 13 different people with different physical characteristics were participant of the recording sessions. In addition some of the participants wore a headband instead of a hat. In some sessions different moving lights were illuminating the Drone Arena. In other sessions, non tracked, non followed users were let into the Drone Arena.
The recorded session lasted for more than 50 minutes ending in almost 45 minutes of usable data. The total file size of the recorded data is 7.5GB.

3.3 Data processing

After the recording sessions, data sources are synchronized, other useful informations are computed and displayed in a video.

Figure 3.4: An image depicting the drone, fitted with passive reflective trackers and a prototype of the user's head tracking hat.

As anticipated in Section 3.1.1, these videos are used to analyze the recording session and search for recording errors similar to the ones described in the same Section. With a thorough manual analysis of the generated videos, annotation of the data are made to indicate the usable parts of a recording. These annotations correspond to the start frame and end frame of the useful part of a bag file.

From these selected sections of recordings, two different types of datasets are created: one made of smaller datasets used for k-fold cross-validation, explained in Section 3.4.3, and one single dataset, already separated in training set and validation set. The first is used for testing the performance of the network on the whole dataset on multiple permutation while the latter one is used for fast testing of changes in the network or in the data.

3.3.1 Synchronizing data sources

As explained is Section 3.1.1, the three main data sources lack synchronization and in some rare cases the three data stream show missing data in the streams; Figure 3.3. To solve both problems, the following solution is implemented.

When we extract the information of a datapoint from a data source, a timestamp is associated to the extracted information. This timestamp corresponds to time of the global timestamps at which messages were received and recorded in the `rosbag`. The tuples are (timestamp, data point) stored in a `pandas DataFrame` one for each `topic`; more on the subject in Section 4.3. This process results in a set of sets of indexed datapoints.

From the resulting data structure, the camera feed is extracted and its datapoints' indexes are used to select the nearest past datapoint in the other two sets. In some cases the poses streams are missing some datapoints and the solution implemented uses the last seen corresponding datapoint; fortunately this problematic cases happen only in the first seconds of recording and are usually discarded later in the process. All other datapoints coming from the poses data streams that are not corresponding to an image frame are dropped.

The reason behind using camera datapoints as reference for synchronizing data streams is because the video feed will be used as input for the CNN model and also because, as said in Section 3.1.1, the camera feed is showing more missing data point resulting in the less numerous stream of data; it wouldn't make sense to have the poses of the drone and user without an image.

3.3.2 Computing new data

Computation of new data, together with the CNN architecture, is the part of the project that has seen more changes during the development.

While building our system, different processed data have been computed. In the current version of the system we compute the user's head relative pose wrt. the frame of reference of the Parrot Bebop 2. The relative pose's variables that we are interested in, are four:

- the x, y and z position coordinates of the relative pose
- the *yaw* orientation of the relative pose.

Pitch and *Roll* are ignored and assumed to be $= 0$.

Computing the relative pose To compute the relative pose of the user's head wrt. drone frame of reference, we will use same concepts explained in Section 1.3.1. First we change the frame of reference of the head's pose from world to drone:

$$^{\text{drone}}T_{\text{head}} = (^{\text{world}}T_{\text{drone}})^{-1} \cdot^{\text{world}} T_{\text{head}} \tag{3.1}$$

To do so we transform the pose from the (position, quaternion) representation to the homogeneous matrix representation, then we apply the Equation 3.1 and obtain the new pose. From the resulting homogeneous matrix we extract the relative position of the user head wrt. the drone frame of reference.

Afterwards, from the two poses' quaternions, we compute the relative Euler angles using the following equation:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan(\frac{2(q_0q_1+q_2q_3)}{1-2(q_1^2+q_2^2)}) \\ \arcsin(2(q_0q_2 - q_3q_1)) \\ \arctan(\frac{2(q_0q_3+q_1q_2)}{1-2(q_2^2+q_3^2)}) \end{bmatrix} \tag{3.2}$$

using the resulting angles, as said before *Pitch* and *Roll* are assumed to be = 0, we compute the relative *yaw* of head wrt. the drone as follows:

$$^{drone}yaw_{head} = yaw_{head} - yaw_{drone} \tag{3.3}$$

then the relative angle values are redistributed between $[-\pi, \pi]$, a typical distribution of these values is shown in Figure 3.5.

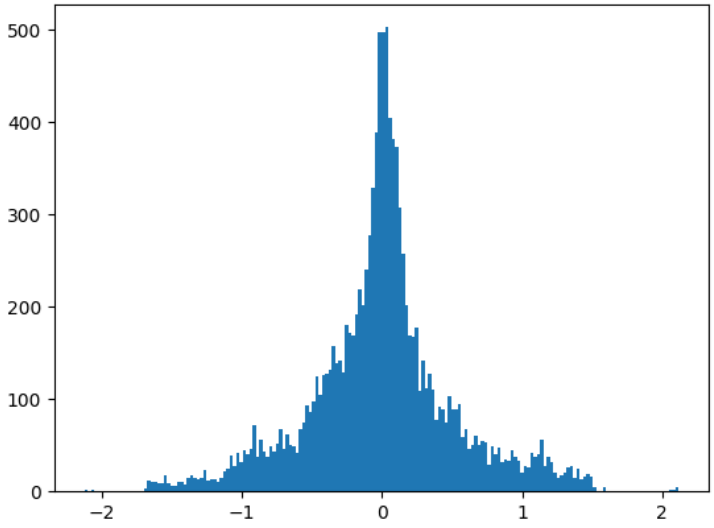


Figure 3.5: Distribution of the relative angle values after processing the data of a bag file. On the x-axis we have radiants, on y-axis frequency.

For the relative pose we don't compute the pitch and roll angles because the video stabilization described in Section 2.1.1 compensates for them and the resulting image is not affected by drone's pitch or roll thus making this variables unusable for our objective.

3.3.3 Debugging the data with video

Using the poses and part of the newly computed data, a video for each bag file is created. In the video multiple data are plotted in addition to the image frame.

During the development, multiple iteration of these videos were made; in Figure 3.6 it is possible to see an old version, while in Figure 3.7 it is possible to see the last iteration of the video structure. Only a smaller part of the newly computed data is used for this video creation.

Video

These videos are used for a purely qualitative analysis of the data.

Using Figure 3.7 as a reference, a frame is built as follows:

- Top-Left plot is a representation of the relative horizontal position of the user in the camera frame as an angle (x-axis) and of the altitude difference (y-axis). The

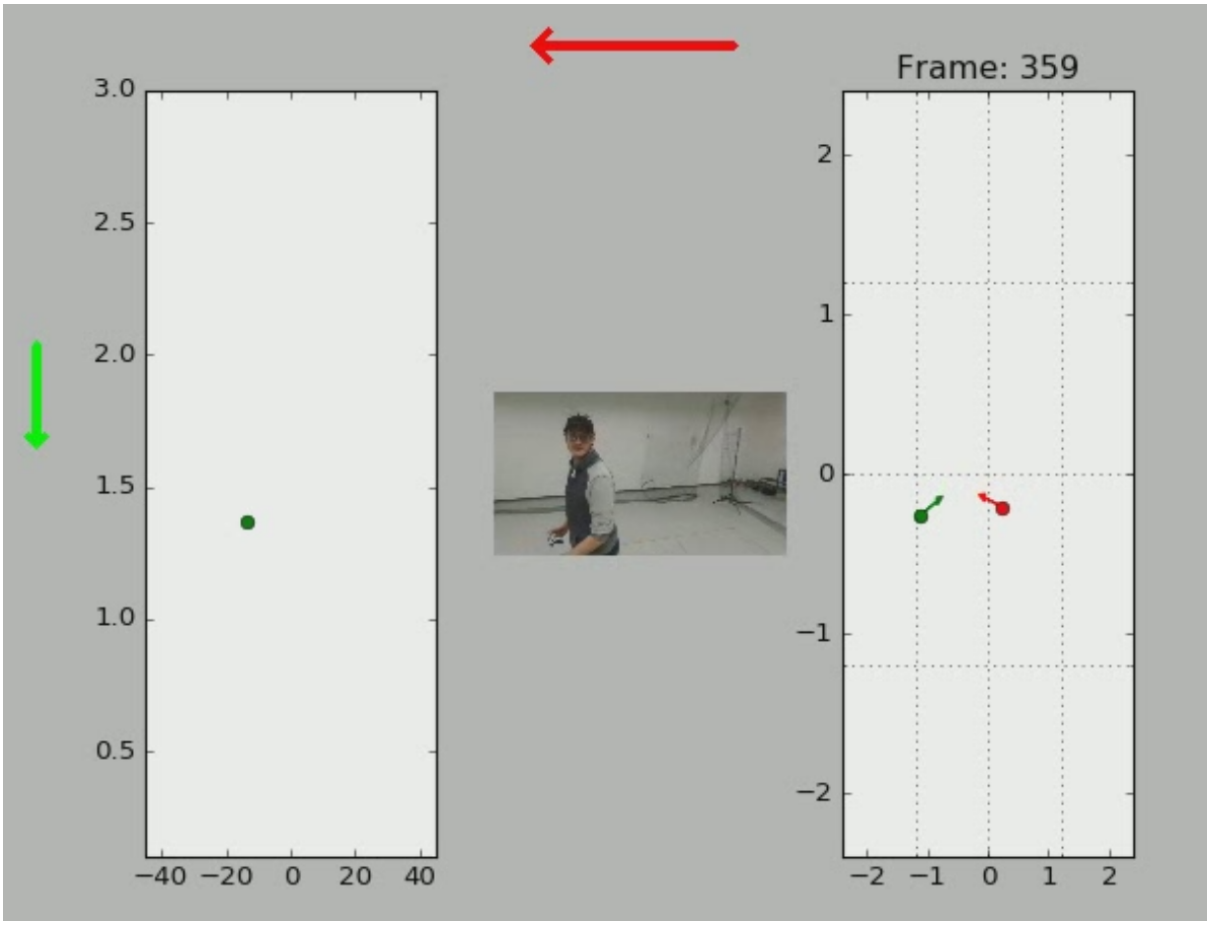


Figure 3.6: An old version of the video described in Section 3.3.3

green dot represents the user's head. This plot can be thought as a front view from the drone reference frame.

- Top-Right plot has on the x-axis the same relative horizontal position but on the y-axis we have the drone-user distance. This can be thought of as a top view of the drone frame of reference, with the green dot as the user and the drone placed at (0,0)[2].
- Bottom-Left is the current frame
- Bottom-right plot represents the top view of the internal drone arena. The green dot is the user and the arrow represents the user's head's yaw (heading). The red dot represent the Parrot Bebop 2 and the red arrow represents the drone yaw. The grid in this plot is corresponding to the tape grid present physically in the drone arena, each square is ($1.2m$ x $1.2m$).
- The two arrows, one horizontal red and one vertical green, are used only for debugging. They represent a binarization of the horizontal angle[3] and distance[4] respectively.

Analysis and cuts

Each video is watched in its entirety and whenever a major recording error or useless data is noticed, an example is shown in Figure 3.8, the first usable frame is annotated as starting frame. Similar annotations are made for last usable frame of a video. These annotations are used to select only the usable portion of the multiple recordings. After the selection, the number of usable frames is 79 739.

3.3.4 Creating the dataset

Before discussing how we create the dataset, a brief introduction to what a training set and validations sets are, is needed.

Training set The training set [4] is a set of multidimensional input vectors associated with their relative true targets values, in case of Supervised training. During the training phase, the model sees the entirety of the training set, learning to predict training targets.

Validation set This is still a set of values and relative targets but instead of using it to optimize model's parameters we use it to check the performance and generalization of the model after training [4]. The model never trains on this set.

Using all the pre-computed data, the creation of the dataset becomes only a matter of scaling the image to $(108, 60)$, the input size for the CNN, and associate to each scaled frame the corresponding tuple of target labels.

[2]Not present in the plot
[3]left = negative angle, right = positive angle
[4]up = distance > 1.5m, down = distance<= 1.5m

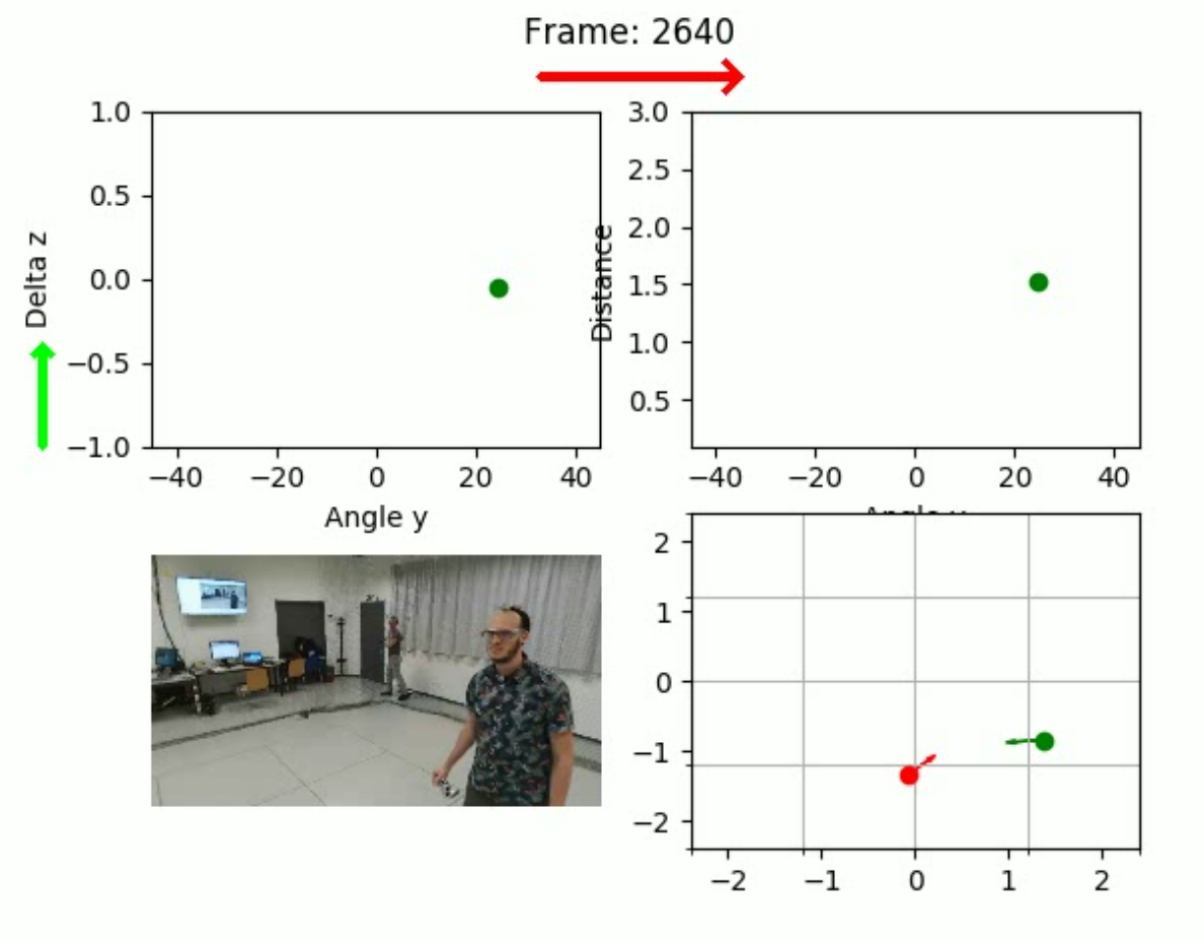


Figure 3.7: A frame of a video used for dataset analysis. A detailed description is present in Section 3.3.3

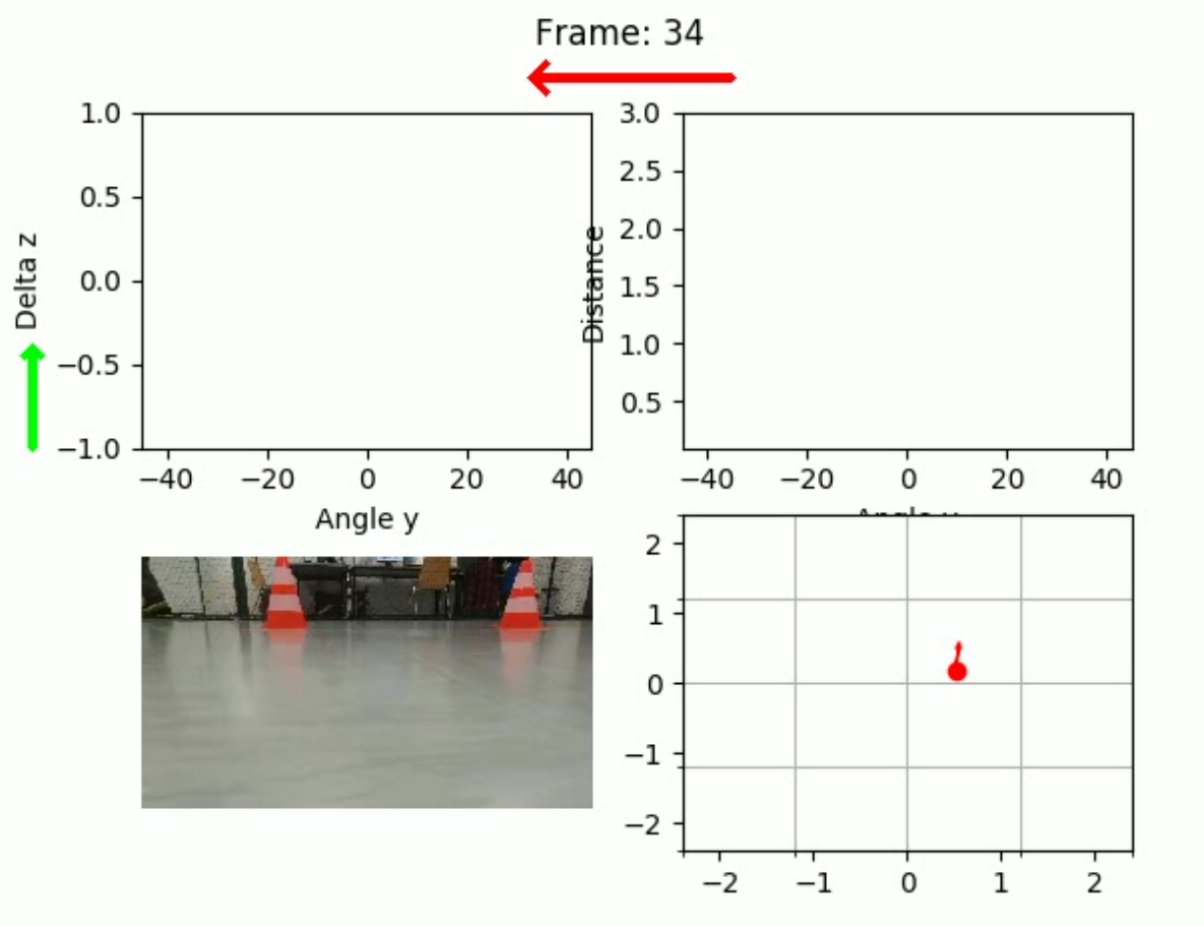


Figure 3.8: A frame of a video that will not be present in the final dataset because the user is not in the OptiTrack tracking area.

Table 3.1: Cross-validation set sizes

Set	Size
0	22992
1	16135
2	11645
3	12529
4	16438

The target label is composed as follows:

- the x coordinate of the relative position of the head wrt. the Parrot Bebop 2;
- the y coordinate of the same point
- the z coordinate
- the head relative yaw between $[-\pi, \pi]$

The actual implementation of the dataset creator, described in Section 4.3, prompts to select what kind of dataset we want to generate. As said at the beginning of Section 3.3, two different types of dataset can be created:

- k-fold cross-validation dataset
- a single, ready to use dataset

The k-fold dataset consists in a folder in which each dataset file `.pickle` is computed and saved from the corresponding bag file. These files are later subdivided into five sets used for the 5-fold cross-validation of the model. To avoid major differences of these sets sizes, the dataset subdivision is hand picked; a bag file is never separated to avoid the presence of the same recording in both train and validation set. The subdivision is shown in Table 3.2 and the resulting set sizes are shown in Table 3.1. For the single dataset a new permutation is used with a $80\% - 20\%$ train-validation division; the size of the train test is 63 726 and the size of the validation is 16 013.

3.4 The model

For this project we decided to use a CNN because, as discussed in Section 1.2.2, a CNN architecture is most suitable for image processing applications. Here we discuss the details of the CNN architecture and the hyperparameters involved.

Our development has followed a spiral process; each iteration followed these steps:

1. design the architecture;
2. test the newly designed architecture with the single dataset divided in 80-20 for train and validation;
3. execute 5-fold cross-validation on the new design thus performing a quantitative and qualitative evaluation of the new architecture;

Table 3.2: Cross-validation dataset subdivision

File	Start	End	Size	Set
1	0	3150	3150	1
2	0	7000	7000	0
3	0	390	390	3
4	0	1850	1850	4
5	0	3840	3840	4
6	0	1650	1650	3
7	58	2145	2087	2
8	63	595	532	0
9	75	1065	990	1
10	50	2089	2039	3
11	0	1370	1370	4
12	470	5600	5130	2
13	40	8490	8450	3
14	50	4450	4400	4
15	0	7145	7145	1
16	0	3500	3500	0
17	0	1400	1400	2
18	0	1300	1300	2
19	0	1728	1728	2
20	220	5070	4850	1
21	0	11960	11960	0
22	222	5200	4978	4

4. perform a qualitative evaluation using videos and on-line regressors controlling the drone.

The first two steps are discussed here, the last two will be discussed in Chapter 5

3.4.1 Designing the network model

The first step of the development spiral almost always coincided with an update of the dataset with more complex target, as already said in Section 3.3.2.

We base our architecture on the CNN architecture presented in [11]. We started out with a very simplified version of their architecture to solve only a subset of the overall problem. We then iteratively redesigned the CNN to solve increasingly complex subproblems.

The final architecture is capable of learning complex enough representations to solve our regression problem and it is structured as follows:

- Convolutional layer of 10 filters $6 \times 6 \times 3$
- Max pooling layer 3×3
- Convolutional layer of 15 filters 6×10
- Max pooling layer 2×2
- Convolutional layer of 20 filters $6 \times 6 \times 15$
- Max pooling layer 2×2
- Dense layer of 256 neurons
- Dense layer of 64 neurons
- 4 neurons for regression

A representation of the final architecture is shown in Figure 3.9, this architecture has 264 689 parameters. All layers have a ReLU activation function associated; the last four neurons(output) have a linear activation function, the activation functions are described in Section 1.2.

3.4.2 Debug and train the model

In this step we test and tune the hyperparameters. We also check that development changes in the architecture are correctly implemented. Hyperparameters were tuned through a process of trials and errors.

Loss function and Optimizer used

Loss function Since the first iterations of the development spiral were focused on a binary classification task, these iterations used MSE as the loss function; the problem solved by these iterations was just a subproblem of the final project scope. Later in the development, the subproblem scope moved from binary classification to regression and we started using MAE instead.

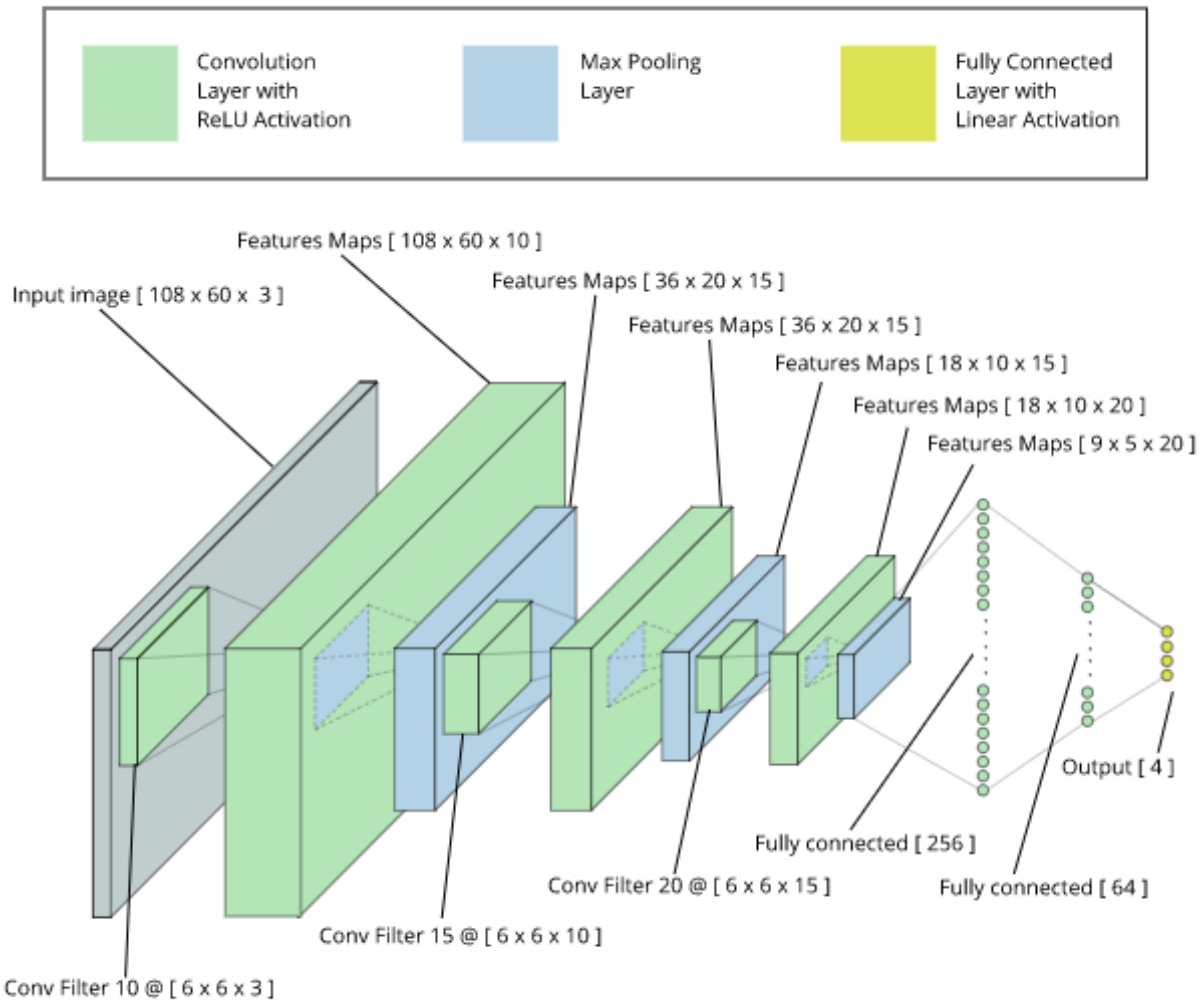


Figure 3.9: Graphical representation of the final CNN architecture.

Mean Absolute Error We have chosen MAE over other loss functions because a direct relation between error and variables values is possible. For example a MAE of 3 for an angle variable (expressed in degrees) corresponds to an error of 3°.
MAE is a measure of the error committed by the model. It can be computed with the following equation:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.4}$$

where $\hat{y}_i$ is the predicted value of the i-th sample, y_i corresponds to the target value, $|y_i - \hat{y}_i|$ corresponds to the absolute value of the difference and n is the number of samples.

Mean Squared Error Even if we don't use MSE as loss function, we still use it as metric function.

MSE is one of the most used loss functions for NN in general. It represents the average squared error between each estimation and the corresponding truth value. It can be computed using the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.5}$$

with the same variable meaning of the MAE.

Adam optimizer The optimizer used in our model is Adaptive Moment Estimation (Adam) optimizer [12].

Adam is a method that computes adaptive learning rates for each parameter. It keeps an exponentially decaying average of past squared gradients v_t.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.6}$$

where g_t is the gradient at timestep t.

In addition to v_t, Adam stores an exponentially decaying average of past gradients m_t, Equation 3.7

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.7}$$

v_t and m_t are estimates of the second[5] and first moment[6] of the gradients respectively. The authors of Adam observed a bias, in initial time steps, towards zero. To counteract this bias, they compute $\hat{m}_t$, Equation 3.8, and $\hat{v}_t$, Equation 3.9, two bias-corrected first and second moment estimates.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.8}$$

[5]Uncentered variance
[6]The mean

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.9}$$

These new moment estimations are used to update the model parameters θ as follows

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{3.10}$$

where η is the learning rate and β_1, β_2 and ϵ are hyperparameters of Adam usually set at 0.9, 0.999 and 10^{-8} respectively.

Mini Batch and Data Augmentation

As part of the training process, we use a mini batch generator and use Adam on the random generated batches.

To create a mini-batch, we use a generator that picks k random samples from the training set. To make the model more robust and reduce eventual symmetry bias present in the dataset, a data augmentation function is applied at all k random samples.

The data augmentation function flips horizontally the image and targets with a 50% probability. More information about the implementation are in Section 4.4.

Early Stopping

Every new model is trained for at least 100 epochs so we can gather data on it. We plot the losses for each target variable both for the validation and train predictions, an example is displayed in Figure 3.10, looking for the moment in which the loss for the training set keeps decreasing while the validation set loss starts to increase; this tells us that the model after this moment will overfits the training data, resulting in worst performance overall. In general we try to avoid that the model overfit the training data so techniques like early stopping are always needed.

3.4.3 K-fold cross-validation

K-fold cross-validation randomly partitions the whole dataset in k subsets; selects $k-1$ samples as training set and the remaining one as validation set then trains and validates the model. This phase is repeated k times, in each turn a different set of the k samples is used as validation and the rest as training. The metrics of the whole cross-validation are averaged.

These results are more significant than a single train and test run because the averaged cross-validation metrics are more robust to biases related to the selection of training and validation set. With this method all observations are used both in training and validation.

In our development we used a 5-fold cross-validation but the partition is made manually in advance to avoid the presence of the same recording in both train and validation set. Each iteration of the cross-validation is run for 100 epochs. Data are saved for every *fold* and for the whole run. Data saved for each iteration comprehend:

- for each target variable, an image composed by four graph, explained in Section 5.1.1;

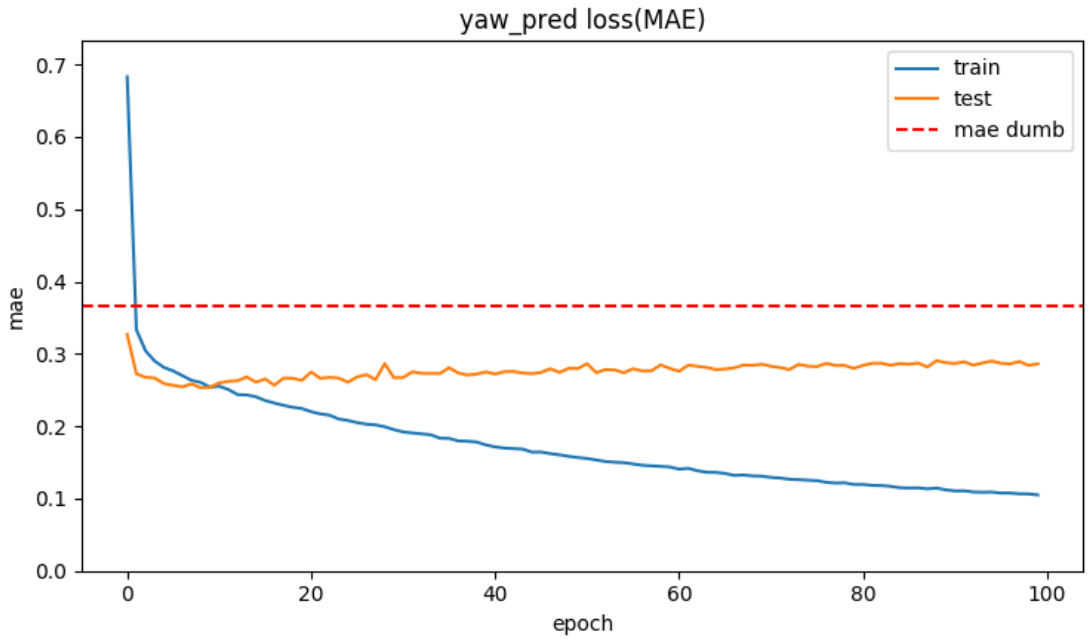


Figure 3.10: This plot is displaying the loss function (MAE) for the yaw prediction both for the train and validation set during 100 epochs of training. The red dashed line corresponds to the dumb regressor loss that always predicts the mean value. This plot is used to explain the concept of Early Stopping.

- a video showing prediction results of the validation set with the 100-th epoch model.
- the CNN graph and weights.

For the whole run we save:

- computation time;
- an image for every target variable composed by two graph showing average results;
- model information;
- metric history for the whole cross-validation.

Given its high computational cost and long run time(more than two hours), 5-fold cross-validation is run only after the model has passed the debugging phase.

Chapter 4

Solution implementation

The system developed is composed by multiple scripts handling different functionalities. Different libraries and framework are used in the code or by the system, the most important are:

- ROS Kinetic Kame
  - `rospy` 1.12.13
  - `rosbag` 1.12.13
- NumPy 1.14.3
- Matplotlib 2.2.2
- OpenCV 2.4.9.1
- `pandas` 0.23.0
- Keras 2.1.6
- Tensorflow-gpu 1.8.0

ROS is already described in Chapter 2. The programming language used for development is Python 2.7.12.
In the following sections we briefly introduce libraries and frameworks and later discuss the implementation of the system.

4.1 Tools

4.1.1 Numpy

Almost in any scripts of the project, the NumPy library is used. NumPy is a core Python library for scientific computing and we use it for almost all array or matrix computation. It is a fundamental Python package for scientific computing.

It provides a multidimensional array object (`ndarray`), various derived objects and a variety of methods for fast operations on arrays, including mathematical, logical, shape manipulation, sorting and others.
The core of NumPy is the `ndarray` object. It encapsulates n-dimensional arrays of

homogeneous data types, with many operations being performed in compiled code for performance.

NumPy is used in many other scientific and non-scientific Python libraries. The two most powerful features of NumPy are vectorization and broadcasting.
Vectorization means the absence of any explicit indexing or looping in the code, while broadcasting indicates the implicit element-by-element behavior of operations, from arithmetic operations to functional operations.

4.1.2 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures. Matplotlib is cross-platform and can be used in Python scripts and in a variety of other Python applications.

Matplotlib makes plotting high quality graphs accessible to anyone. With Matplotlib it is possible to generate a wide variety of types of plots from histograms to scatter plots with just a few lines of code.

Matplotlib uses a MATLAB-like interface. Power user have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions similar to MATLAB one.

4.1.3 OpenCV

Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning library. Like ROS, OpenCV is a BSD-licensed product.

The library is composed by more than 2500 optimized algorithms, including both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used in a variety of ways from detecting and recognizing faces to object identification or object tracking.

The library has an immense user community counting more than 47 thousand people. The library is used extensively in companies and research groups.
OpenCV has helped robots navigate and pick up objects at Willow Garage, the company behind the diffusion and development of ROS.
OpenCV is available for different OSs and in different languages, it has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS.

4.1.4 pandas

pandas[1] is an open source community supported, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for Python. pandas is still in active development and since it has seen a growth in adoption, on the official documentation multiple guides are present to help shifting from another environment or language like R.

[1]The name is written in lowercase letters.

4.1.5 Keras and TensorFlow

Keras Keras is a high-level Neural Network API, written in Python capable of running on top of TensorFlow, CNTK[2], or Theano. It is designed to allow fast experimentation and validation. It focuses on being user-friendly, modular, and extensible. By being a high-level API Keras can run multiple NN architecture, from CNN to RNN and combinations of these. The back-end that we use in conjunction with Keras is TensorFlow, an open source software library for high performance numerical computation that uses data flow graphs.

TensorFlow TensorFlow is one of the most used library for machine learning applications such as neural networks. It can run on multiple CPUs, GPUs and, since 2016 Google's announcement, it can run on TPUs[3]. TensorFlow is scalable and multi-platform.

The core elements of the library are tensors and the computational graph. Tensors can be interpreted as multidimensional array, e.g. a Rank-1 tensor is an array of real numbers. The computational graph consists of nodes and edges. Nodes are operations receiving and producing zero or more tensors, edges are connections between nodes. The version of TensorFlow used in this project runs on GPU.

4.2 Dataset analysis scripts

The analysis discussed in Section 3.1, is done using the functionalities of the script `pre_processing.py`.
Through a simple command line interface, the script exposes multiple functionalities:

- compute user-drone mean distance of all the recorded data;
- create an old version of the dataset (using a multi thread approach);
- create a video for each bag file.

Being one of the first script implemented, the program is pretty inefficient both in term of speed and in term of primary memory occupation. Nonetheless the video and mean distance functionalities are often used to have a better understanding of the dataset.

The script uses different libraries. For video creations it uses a combination of `cv2`, an OpenCV implementation for Python, and `matplotlib` the most famous plotting library for Python, the script is displayed in Listing B.3.

4.3 Dataset creator scripts

For dataset creation, described in Section 3.3, the script `fast_pre_processing.py` is used. This script is heavily optimized and can elaborate the whole dataset in less than 30 minutes.

[2]Microsoft Cognitive toolkit
[3]Tensor Processing Units, an Application Specific Integrated Circuit (ASIC) built for machine learning problems running on TensorFlow

The improvements in computational speed and memory occupation, if compared to the almost 4 hours needed by `pre_processing.py` to compute the same amount of data, are due to two main reasons:

- code refactoring and function optimization;
- swap from using different `numpy` arrays and Python lists to `pandas DataFrames` .

As the previous script, even this one presents its multiple functionalities through a simple command line interface. These functionalities are: create cross-validation dataset or create the single instance of the dataset, as described in Section 3.3.4. Using the Python library `multiprocessing` the cross-validation dataset can be created using the multi-processing implementation.

The code is shown in Listing 4.1 and the function mapped is in Listing B.7.

Listing 4.1: Extract that shows multi-process dataset creation, where `Pool` is a method of `multiprocess` library

```
pool = Pool(processes=4)
pool.map(bag_to_pickle, files[:])
pool.close()
pool.join()
```

In both uses of `fast_pre_processing.py` data extraction from bag files must be executed. In Listing 4.2 is present a section of a method that reads ros messages from a `topic` and fills a `pandas DataFrame`.

Listing 4.2: Extract that shows how to read messages from a topic of a ros bag file

```
# [....]
h_id = []
h_v = []
for topic, hat, t in bag.read_messages(topics=['/optitrack/head
    ']):
    secs = t.secs
    nsecs = t.nsecs
    h_id.append(time_conversion_to_nano(secs, nsecs))
    pos_rot_dict = (lambda x, y: {'h_pos_x': x.x,
                                  'h_pos_y': x.y,
                                  'h_pos_z': x.z,
                                  'h_rot_w': y.w,
                                  'h_rot_x': y.x,
                                  'h_rot_y': y.y,
                                  'h_rot_z': y.z})(hat.pose.
                                     position, hat.pose.
                                     orientation)
    h_v.append(pos_rot_dict)
head_df = pd.DataFrame(data=h_v, index=h_id, columns=h_v[0].
    keys())
# [....]
```

Read messages from a `rosbag` To be able to read messages from a topic, first we need to understand the messages types. In Section 4.3.1 we describe `PoseStamped` message and `CompressedImage`.

As part of both dataset creation, Section 3.3, and video creation for data analysis, Section 3.1, processing of the data is needed.

Here we analyze the latest implementation of the `processing` method of `fast_pre_processing.py`.

This method, called at line (12) of Listing 4.2, is the main hub of data processing. From line (11) to (16), different arrays are initialized or extracted from the dataframe dictionary `bag_df_dict`. After that, in a `for` loop for each datapoint of the recording, we execute the following steps:

- find the nearest poses to the datapoint image frame, from line (18) to (21), using the Listing B.1;
- at line (22) the image raw data are extracted from the `pandas` dataframe;
- we change the frame of reference of the head's pose at line (23), using Listing B.2;
- from line (24) to (28) we compute, with the needed steps, the relative yaw;
- the last three lines create the data structure for the CNN and append it to the data vector.

4.3.1 ROS messages

Message `PoseStamped` To represent the users' head's odometry or the Parrot Bebop 2 's odometry at a certain time stamp, a `PoseStamped`[4] message is used. The `PoseStamped` message is a message composed by a `Header` message and a `Pose` message.

The `Header` message contains three variables:

- a integer indicating the sequence ID
- a time stamp value expressed in seconds and nanoseconds
- a string indicating the frame related to the pose.

the `Pose` message instead is composed by two other ROS messages a `Point` and a `Quaternion`. These messages are composed as follow:

- `Point`:
  - `x`: a float number indicating the x coordinate in the 3-D space;
  - `y`: a float number indicating the y coordinate in the 3-D space;
  - `z`: a float number indicating the z coordinate in the 3-D space;

[4] `http://docs.ros.org/melodic/api/geometry_msgs/html/msg/PoseStamped.html`

- `Quaternion`:
  - `x`: a float number indicating the x coordinate of a quaternion;
  - `y`: a float number indicating the y coordinate of a quaternion;
  - `z`: a float number indicating the z coordinate of a quaternion;
  - `w`: a float number indicating the w coordinate of a quaternion.

Message `CompressedImage`

The camera information is a ROS message of type `CompressedImage`[5]. This message is composed of:

- an Header;
- a string that represents the format of the image;
- the compressed image buffer.

4.4 Model scripts

The CNN model implementation and utilization are distributed across four scripts:

- `model_creator.py`
- `keras_train.py`
- `keras_crossvalidation.py`
- `keras_ros_node.py`

These scripts rely heavily on Keras library with Tensorflow as back-end. These two libraries are described in Section 4.1.5.
The complete code of the first three scripts can be found in Appendix B, `keras_ros_node.py` is too long to be included but can be found on GitHub[6].

In `model_creator.py`, Listing B.6, we define the architecture of the CNN, the data augmentation method and the batch generator. This file is imported in the first two scripts to instantiate the model and generate the batches used for training.

`keras_train.py` is used to train and test the model on the single dataset. This script is used when testing new targets or changes to the architecture. Usually plots and data from this script are not saved but only showed on screen. To do a qualitative evaluation on the changes, `keras_train.py` relies on another Python file: `tool_to_plot_data.py`[7].

Cross-validation is implemented in `keras_crossvalidation.py`. Here we apply the code that we have tested in `keras_train` and let it run on multiple[8] permutations of the

[5] `http://docs.ros.org/api/sensor_msgs/html/msg/CompressedImage.html`
[6] `https://github.com/Mardox91/DeepDrone`
[7] This script is too long to be inserted but it is available at the GitHub`https://github.com/Mardox91/DeepDrone`
[8] Five in our implementation

dataset while saving metrics, running informations, the plots and the models trained.

In both `keras_train.py` and `keras_crossvalidation.py` after training and testing the network, a dumb regressor is run and both qualitative and quantitative evaluation techniques are applied to the results, the implementations of the dumb regressor and evaluation are discussed in Section 4.5. Model from the previous scripts are saved in the `h5py` format, these files are used for post run evaluation.

`keras_ros_node.py` is a on-line test for already trained models. Loading the whole model, previously saved, allows us to test it in a real environment. By converting the output of the model into a ROS message, it is possible to let the CNN model to control the Parrot Bebop 2 inside the Drone Arena. This script is launched as a ROS node.

4.5 Evaluation scripts

Most methods for evaluations are used in other scripts but are implemented in different files.
`tool_to_plot_data.py` contains multiple custom methods used by other scripts to plot results both as images or as videos.
Another important component for evaluation is `dumb_regressor.py`. A dumb regressor is a regressor that always predicts the mean value of a certain target.

The script implements the class `DummyRegressor` of the library `sklearn`, used at line (20) of Listing B.9. From line (23) till line (36) we generate metrics for the regressor. The reason of using this type of regressor for evaluation will be explained in Chapter 5.

The only stand-alone evaluation script is `post_flight_analysis.py`. While running `keras_ros_node.py` we would record all messages on a bag file, `post_flight_analysis.py` is made to analyze those bag file and produce a different video used for qualitative evaluation. More on the subject in Chapter 5.

Chapter 5

Evaluation

In this chapter we discuss the evaluation of our solution, continuing the description of the spiral development introduced previously. The evaluation of the project has both an off-line and on-line part.

With off-line evaluation we identify the evaluation of prediction made on pre-recorded data. On-line evaluation is done on prediction made by the model while it is controlling the drone in a real-robot scenario.

While the on-line evaluation is only a qualitative evaluation, the off-line evaluation is divided in qualitative and quantitative. The metrics used are described in Section 3.4.2.

An important component of the evaluation is the *dumb regressor*.

Dumb Regressor A dumb regressor is a regressor that always returns the mean value of a variable. Its prediction is compared with the true values of the variable and using the same metrics of the real model, an error is produced.This value corresponds to the minimum performance that the real model must achieve.

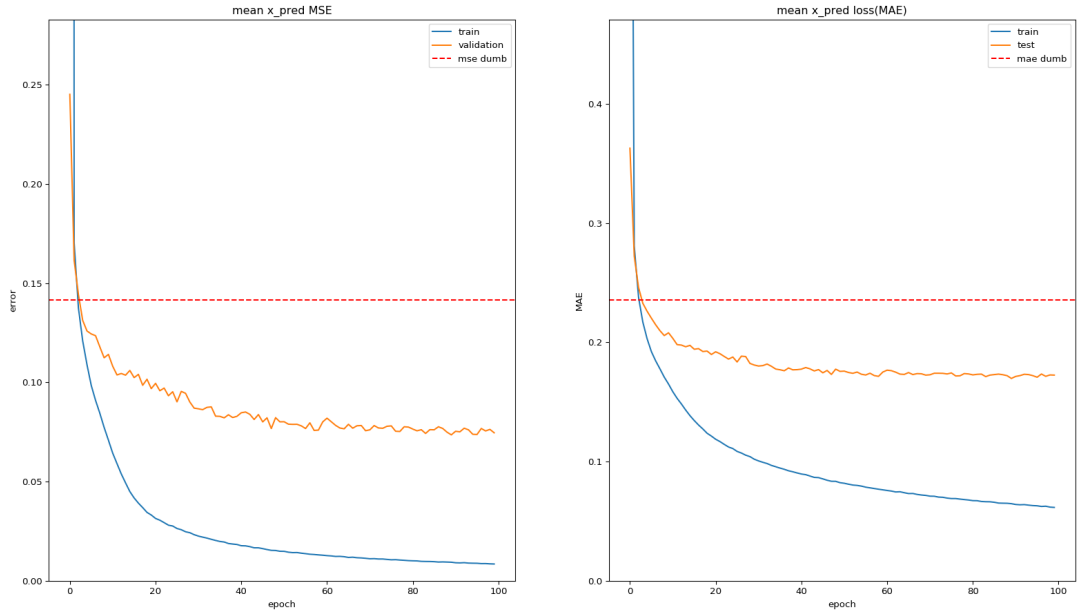
5.1 Off-line evaluation

For the qualitative off-line evaluation we analyze videos composed using video feed, target values and predicted values; a frame is visible in Figure 5.2. For the quantitative evaluation we gather data from MSE and MAE metrics later plotted on multiples graphs.

5.1.1 Quantitative evaluation

From the cross-validation data we can perform a quantitative evaluation of the model. For each target variable we average the metrics values across cross-validation folds and multiple graphs, shown in Figures 5.1, are used to represent different point of view on the target variables' predictions.

By looking at these graphs we can say that the CNN model learned enough to perform better than the dumb regressor, its error is the red dashed line in Figure 5.1a, both on the training set and validation set. Further analysis is done using the graphs in Figure 5.1c, these graphs are made one per fold and one for each variable; the variables are described



(a) x MSE and MAE plots over 100 epochs

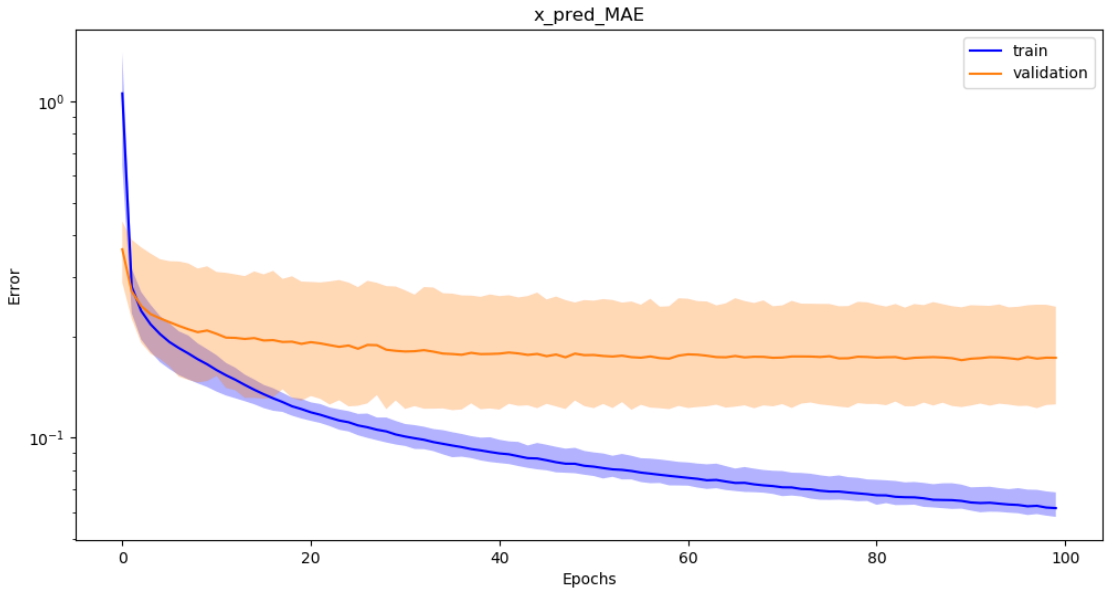
in Section 3.3.4.

These graphs show data only for a single target variable and are composed by 4 plots:

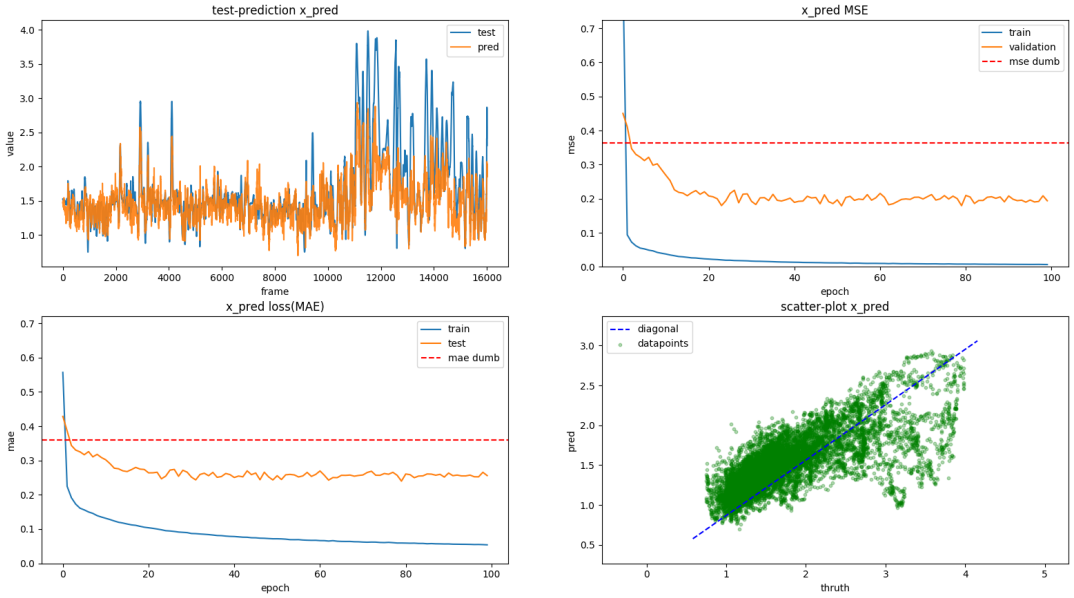
- top left graph is a representation of the truth and prediction values relative to the targets of the validation set along all datapoints;
- top right and bottom left are the MSE and MAE metrics for the target variable in a fold;
- bottom right plot is a scatter plot of the validation set values for the selected target variable with the truth on the x-axis and prediction on y-axis.

A model that makes good predictions produces a scatter plot squeezed along the diagonal line. From the scatter plot in Figure 5.1c we can affirm that, even if some outliers are not well recognized by the model, most of the datapoints are along the diagonal.

The results just described are confirmed for every target variable by the graphs in Figures A.1 and A.2. If we average the metrics along 100 epochs, we obtain the data in Table 5.1. This gives a wrong representation of the results: almost all the validation averaged metrics have lower values than the train averaged metrics, while in reality the training metrics have always lower values than the validation metrics. Using the plot in Figure 5.1b we can easily understand that the prediction error for epoch 0 is very high and affects the averages, this value is so high that we have to plot the data with a logarithmic scale for the y-axis. Excluding the first epoch, the averaged metrics result in an expected behavior with train metrics values lower than validation ones; Table 5.2 and Figure 5.1d.



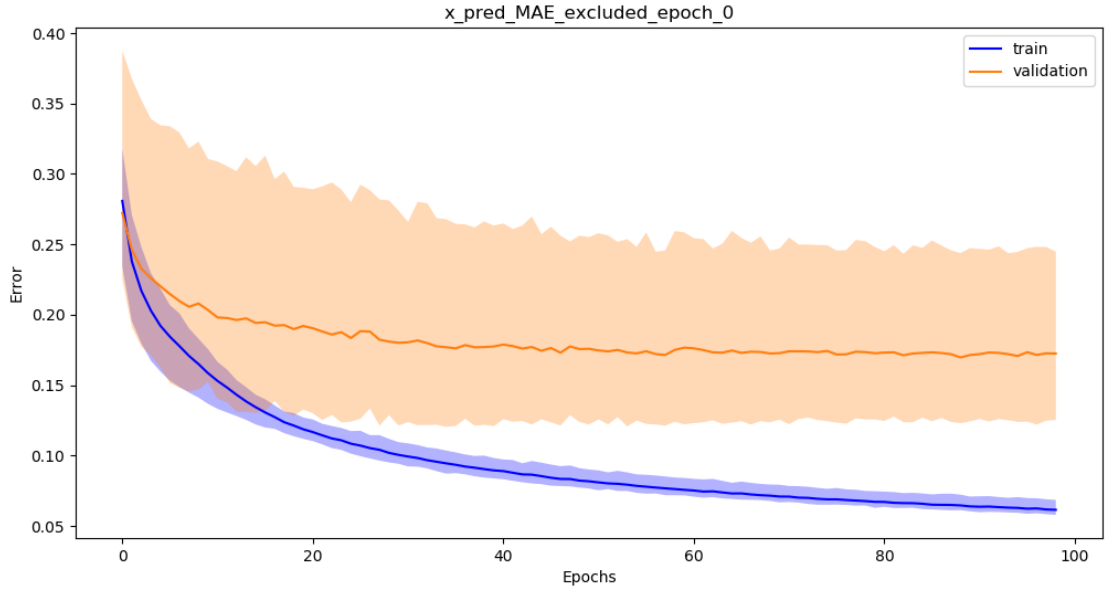
(b) x MAE plots over 100 epochs for the train and validation sets. The transparent area is defined by the maximum value of one of the folds and the minimum value of one of the folds. The line is the mean fold result.



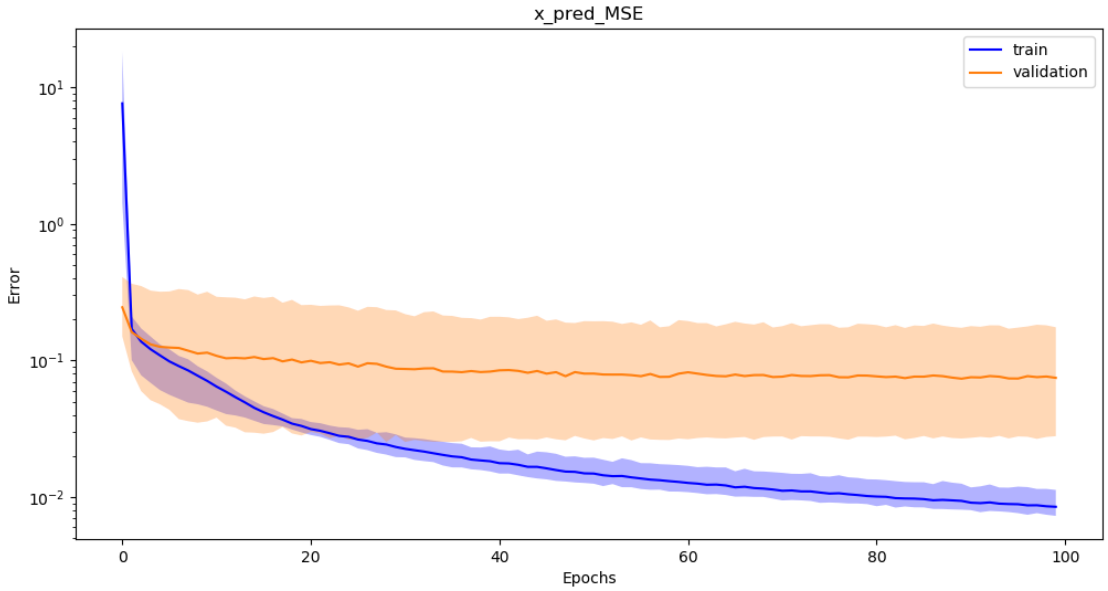
(c) A figure with 4 plots of a fold for variable x. These graphs are described in Section 5.1.1

Table 5.1: Mean Full Results across 5-fold cross-validation

	x		***y***		***z***		***yaw***	
	train	validation	train	validation	train	validation	train	validation
MAE	0.107	0.184	0.080	0.100	0.088	0.117	0.186	0.291
MSE	0.102	0.089	0.086	0.026	0.072	0.026	0.135	0.177



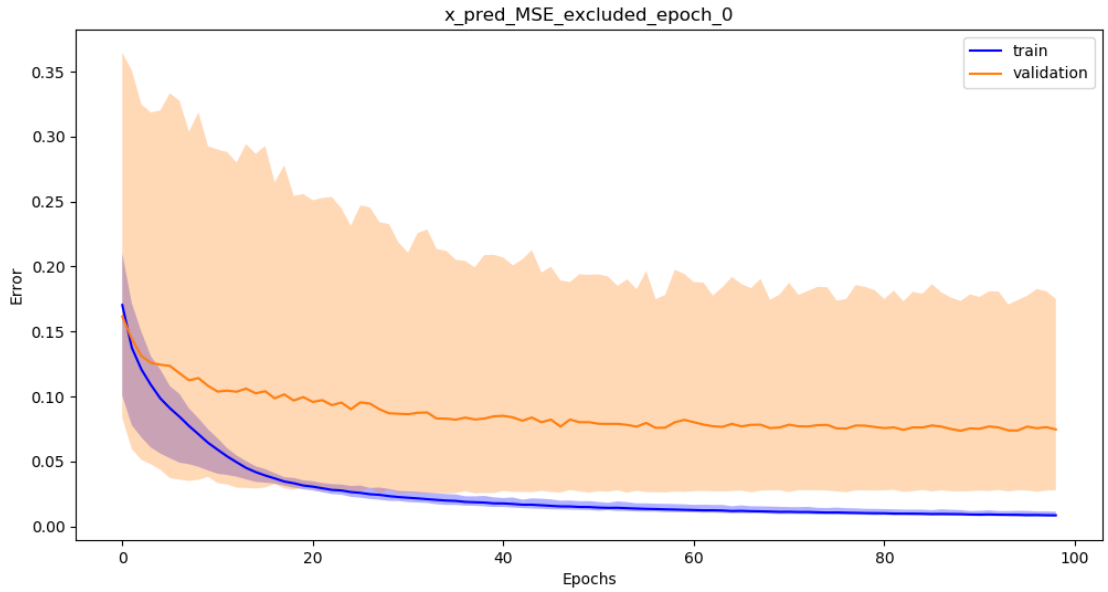
(d) x MAE plots over epochs 1 to 100 for the train and validation sets. The transparent area is defined by the maximum value of one of the folds and the minimum value of one of the folds. The line is the mean fold result.



(e) x MSE plots over 100 epochs for the train and validation sets. The transparent area is defined by the maximum value of one of the folds and the minimum value of one of the folds. The line is the mean fold result.

Table 5.2: Mean Results across 5-fold cross-validation excluded epoch 0

	x		***y***		***z***		***yaw***	
	train	validation	train	validation	train	validation	train	validation
MAE	0.097	0.183	0.070	0.098	0.078	0.116	0.177	0.290
MSE	0.026	0.087	0.014	0.026	0.013	0.025	0.087	0.176



(f) x MSE plots over epochs 1 to 100 for the train and validation sets. The transparent area is defined by the maximum value of one of the folds and the minimum value of one of the folds. The line is the mean fold result.

Figure 5.1: Graphs for the variable x. The line color indicate the set predicted blue = train, orange = validation, red dashe = dummy regressor on validation.

5.1.2 Qualitative evaluation

The qualitative evaluation is done through videos illustrating predictions on the validation set, one for each fold, represented as in Figure 5.2.

Video

The video is composed by three areas: on left side a representation of the top view of the drone FOV , a central plot representing the height truth(green dot) and prediction(blue dot) and the right side with the view of the input signal for the CNN with a 4x scale. The left plot is composed by:

- a top view image of the drone positioned at the $(0, 0)$ point of the drone frame of reference;
- vertical and horizontal scales in meters;
- a representation of the drone FOV after image-stabilization;
- the truth value for the user position in the drone's FOV, represented as a green circle (position) and arrow (yaw);
- the predicted value for the user position in the drone's FOV, represented as a blue circle (position) and arrow (yaw).

On the top section of a frame, we write the values of the prediction and truth values of all variables at that timestep. Using these videos we can highlight instances of correct

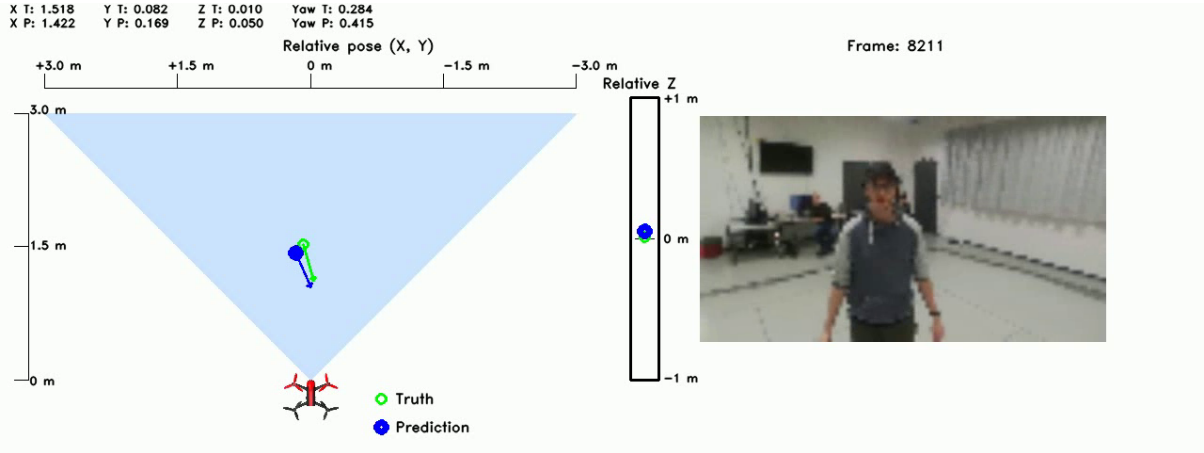


Figure 5.2: A frame of a off-line qualitative evaluation video

behavior from the CNN model and instances of incorrect behavior usually linked to outliers situation present in the data.

Now we analyze six situations coming from the cross-validation videos.

Correct behavior

Figure 5.3a In the frame depicted, the user is crouching while looking at the drone. The CNN correctly predicts position, relative altitude and heading of the user.

Figure 5.3b Here, the user is crouching while another person is passing by. The model correctly predicts all targets and ignores the bystander.

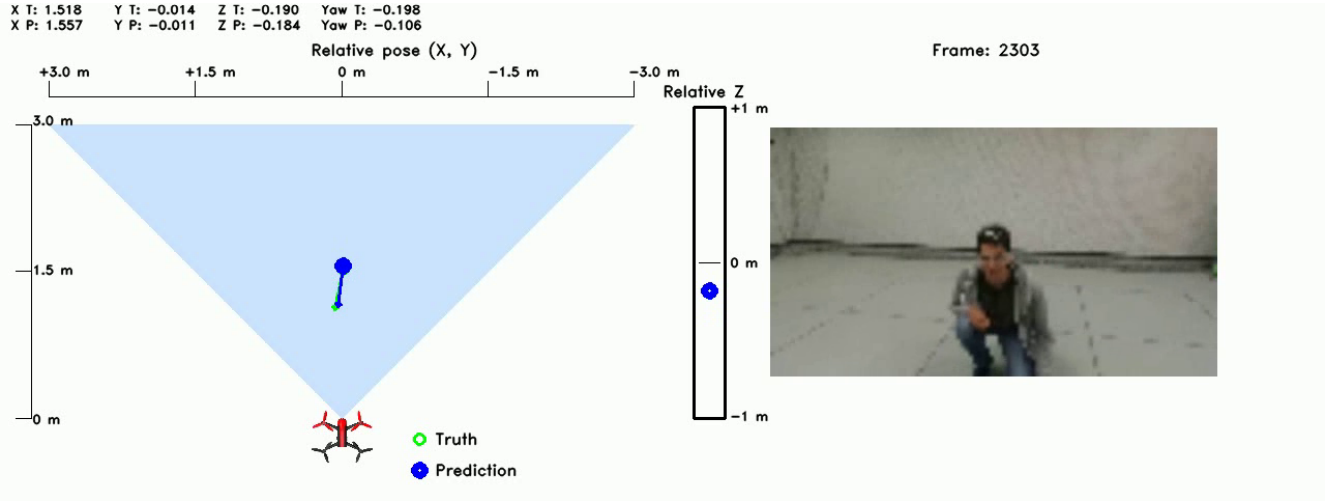
Figure 5.3c In this situation, the network correctly predicts an extreme case in which the user is facing away and is not positioned in the center of the image. Cases like the one discussed are extreme and rare, this is due to human behavior, it is very unlikely that a human would face away from a fling drone at a near distance without a direct instruction to do so. This bias is noticed even more in the on-line evaluation.

Incorrect behavior

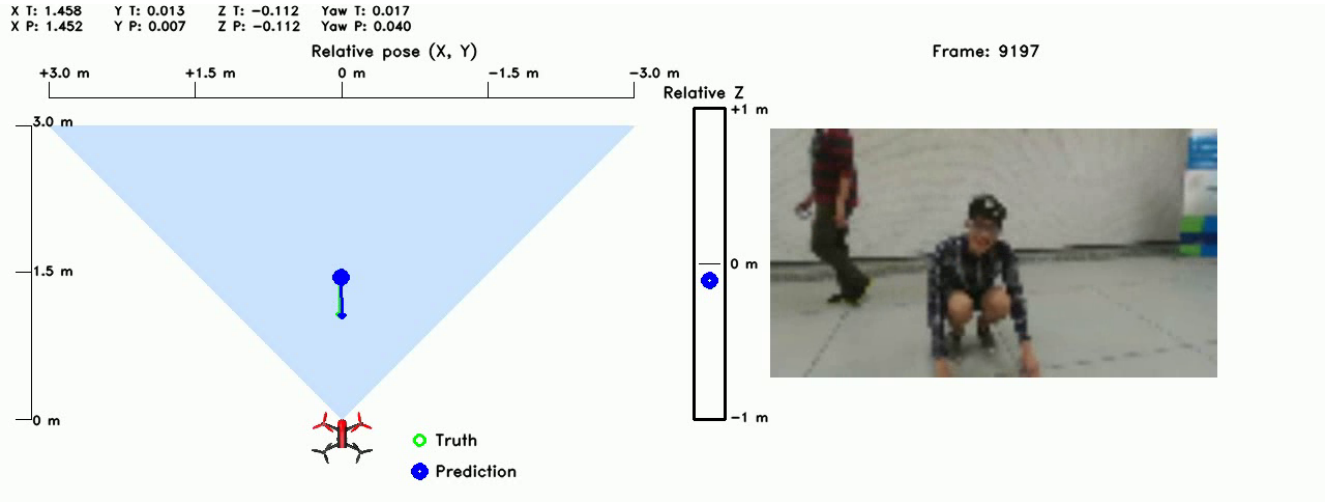
Figure 5.4a In this picture we have a situation similar to the one depicted in Figure 5.3c but here a change in illumination and a different movement of the user result in a wrong prediction.

Figure 5.4b This frame shows one limitation of the network, high illumination beams directed to the camera. In the video sequence around this frame the network continues to correct its prediction without success.

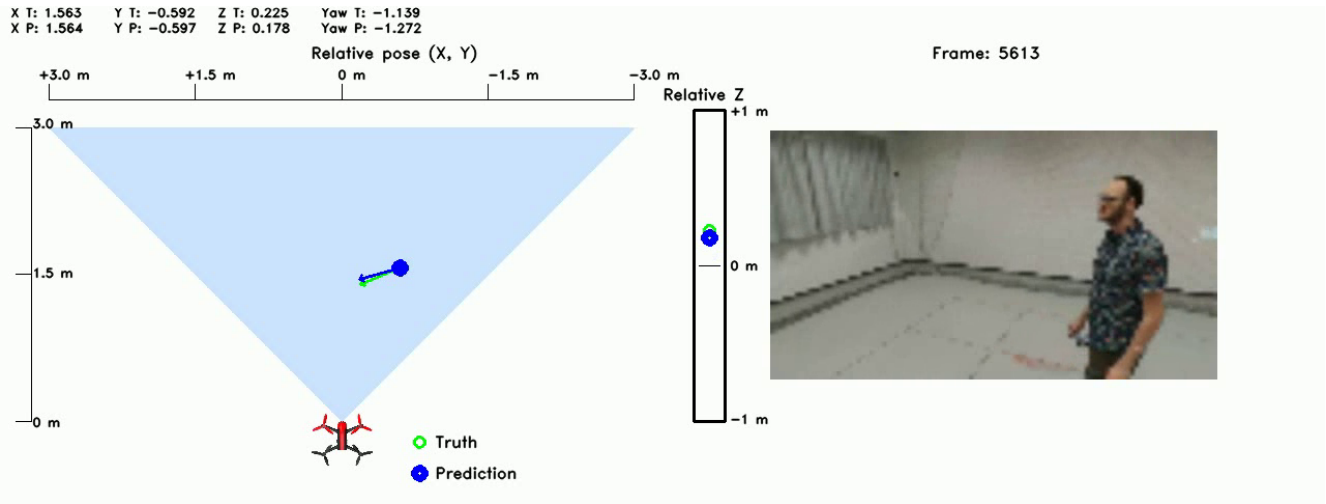
Figure 5.4c In this situation the user movements span over wider area than what the CNN model has seen during the training. This results in cautious prediction by the model still resulting in an incorrect behavior. Still, in a on-line test the network would slowly move the drone thus correcting the prediction over time.



(a) An example of correct behavior

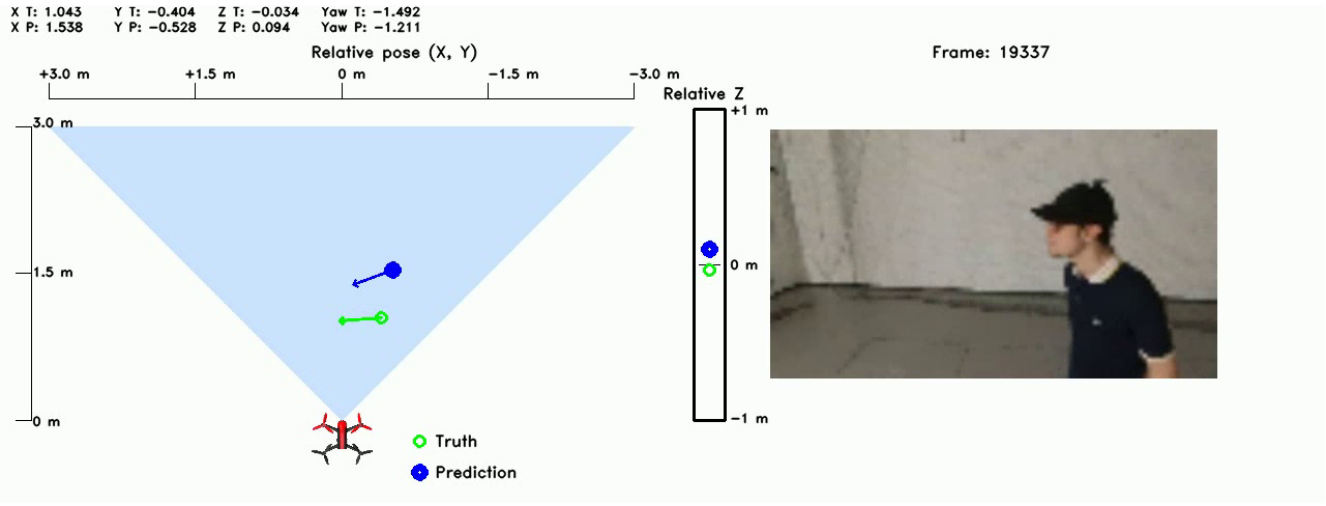


(b) Another example of behavior expected

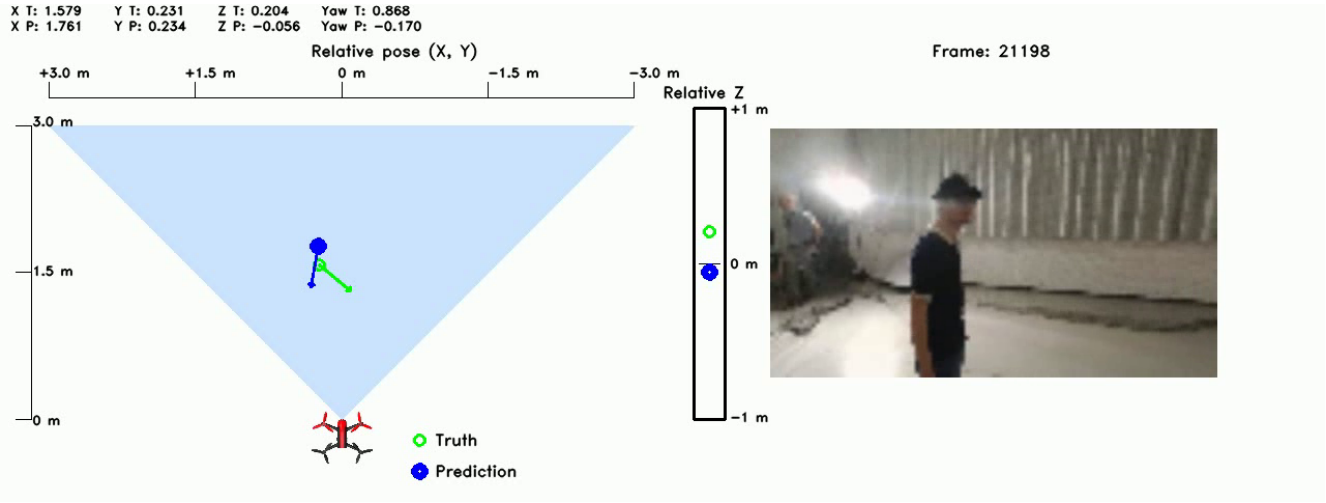


(c) An example of correct behavior in an extreme case

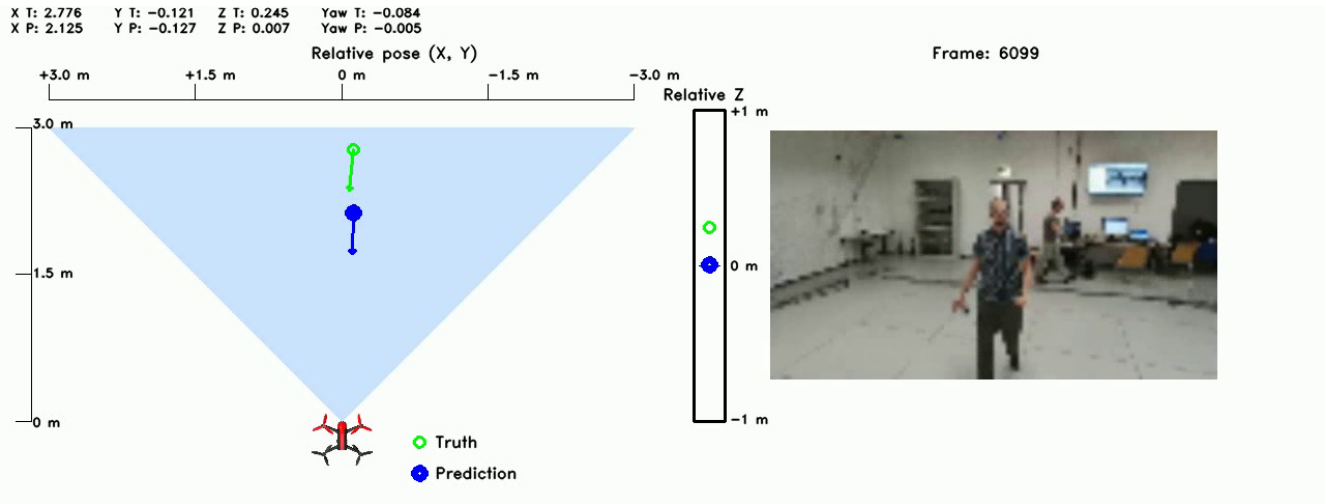
Figure 5.3: Examples of correct behavior



(a) An example of incorrect behavior.



(b) A frame showing an incorrect behavior



(c) Another example of wrong prediction

Figure 5.4: Example of incorrect behavior

5.2 On-line evaluation

The on-line evaluation is a qualitative evaluation in which we tested a trained model running on a ROS node posting target pose and head relative pose used to control the drone.

While running, all data from all topics are recorded and after the flight a video is generated. An example of the video used for qualitative evaluation is illustrated in Figure 5.6 and an illustration of the functioning of the CNN demo is in Figure 5.5.

5.2.1 Qualitative evaluation

The videos are composed as the previous ones with the addition of a red circle with red arrow representing the target pose of the drone[1].

With an on-line test is more difficult to evaluate CNN model behavior. This is due to two main reasons: the drone dynamic and the controller. The first problem is related to the physical limitation of the drone flight, it cannot replicate sudden changes as fast as the network predictions change. The second problem is related to the fact that the network predicts a pose, this pose is used to generate a target pose for the drone that is sent to the drone controller. This controller has some parameters related to maximum acceleration or speed. With a less restrictive parameters on speed and acceleration, if the user stays in place, the drone ends up in a sequence of movements similar to a pendulum oscillation continuously trying to reach a target pose. If the parameters are more strict on the drone movement, the result is a slower, more stable and smoother flight, sometimes resulting in errors in prediction due to the inability of the drone of keeping up to the user movement.

Examples

Figure 5.7a, 5.7b In this cases the drone is capable of following user movement and the prediction results are precise.

Figure 5.7c The user is crouched and the drone is slowly decreasing altitude, the model predicts correctly the target altitude but fails, due to drone movements while descending, to predict the user position.

Figure 5.7d, 5.7e In these two scenes, the drone is ignoring the user heading. Usually after a mild try to face the user, as in first Figure, as soon as the user is more in the center of the FOV the model completely ignores the user heading, second Figure. This is due to the bias present in the recording. User during recording would rarely face away a flying drone that is able to move very fast. The results of such bias are evident in situations like the one just described.

[1]The drone target pose is not present in the height bar because it would simply correspond to the prediction circle

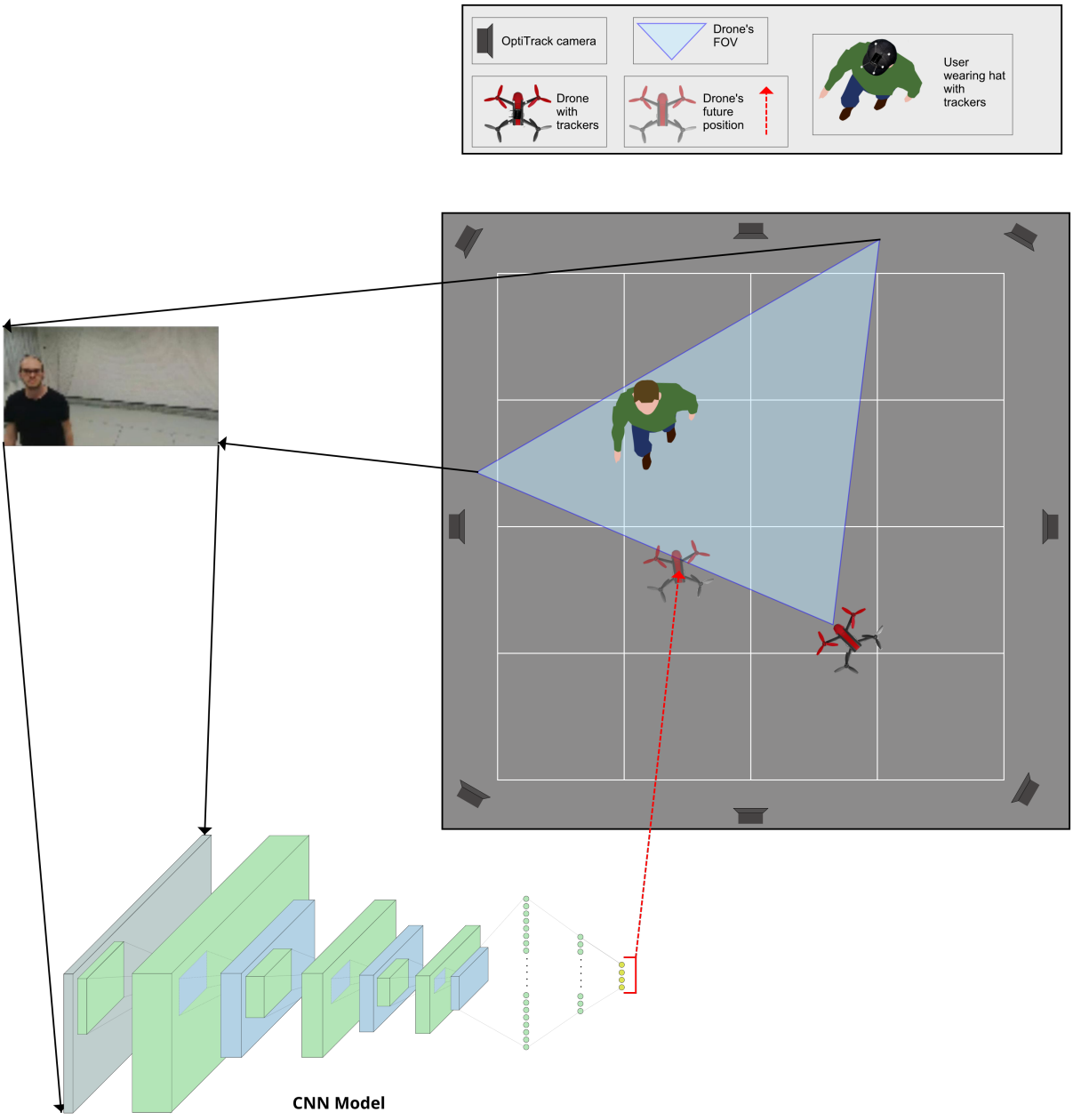


Figure 5.5: An illustration of the CNN on-line demo.

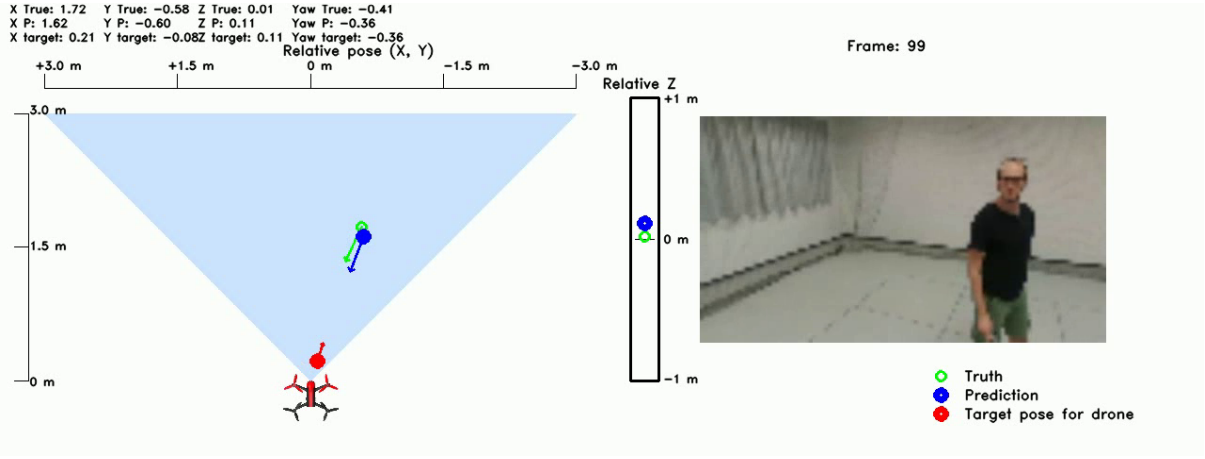
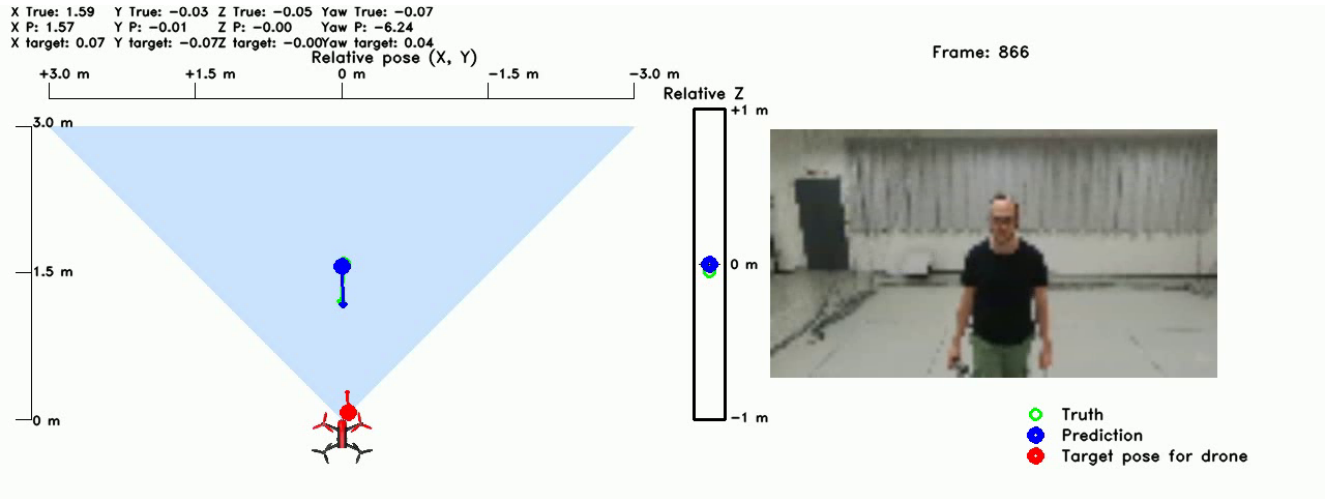
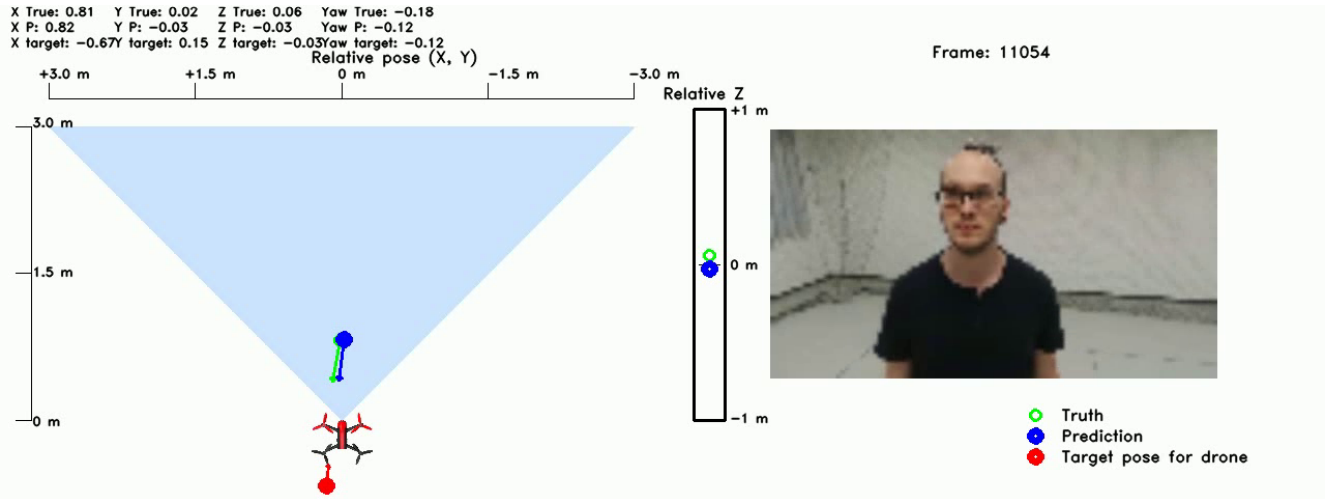


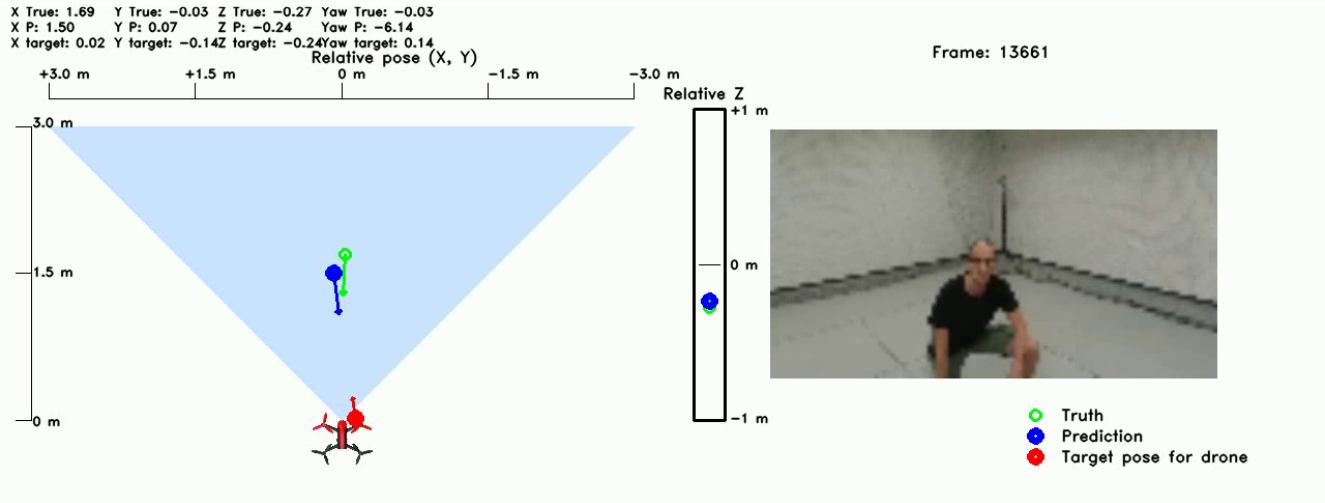
Figure 5.6: A frame of a post flight evaluation video.



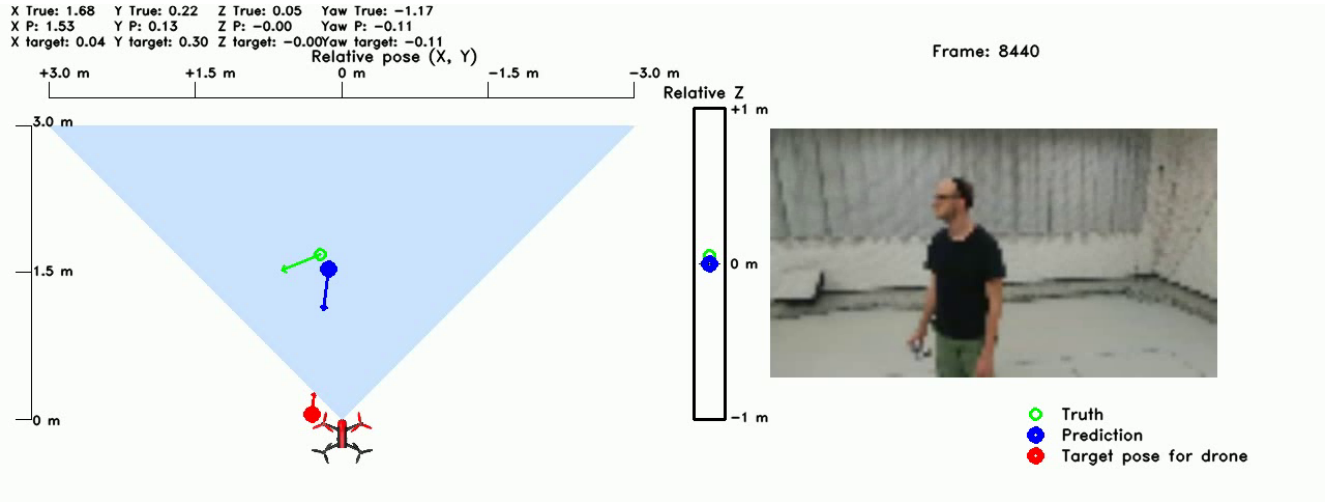
(a) An example of the on-line test showing correct behavior.



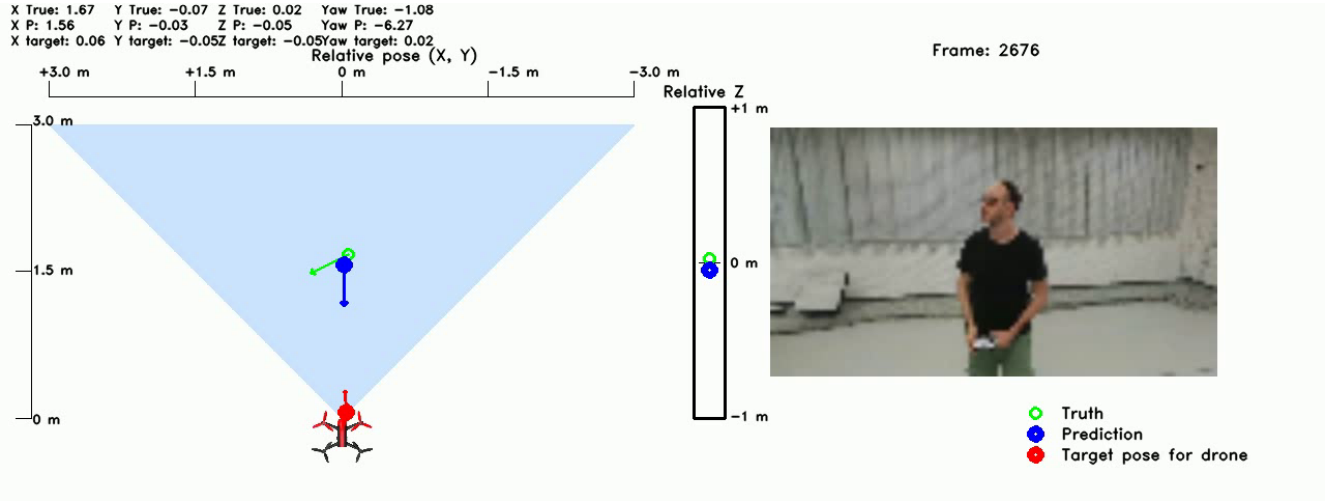
(b) Another example of correct behavior even in close proximity



(c) Here the model presents uncertainty of prediction



(d) The model ignores the user heading if it is in a central position



(e) Another example of the CNN ignoring partially the heading.

Figure 5.7: Examples of model behavior in a on-line test

Chapter 6

Conclusions and future work

6.1 Conclusions

The objective of this project is to develop a CNN model capable of imitating a precise drone controller using only the drone's front facing camera instead of a MoCap system. To reach this objective first we analyze the data coming form the sources available, selecting the ones which are useful as input and target for the CNN model to predict the relative user's pose. We record raw data during 22 recording sessions and from these data we compute the dataset used to train the CNN.

We measure the performance of our solution during training and synthetic test with two metrics and video analysis. During real life test we asses the model performance using post flight video analysis. With the evaluation results we can claim that our solution has very good performance during validation test, as showed in Section 5.1.1.

The off-line evaluation tells us that the model has learned to predict with a very good accuracy the lateral(along y-axis) and vertical(along z-axis) positions of the user in the camera frame, MSE$<$ 0.027, and has good accuracy with minor prediction errors for user's distance along x-axis,MSE$<$ 0.087, and user's heading direction,MSE$<$ 0.176.

The model shows very good performance when controlling the drone during real life tests:

- the model produce prediction in real time at a stable frequency of 10Hz;
- it can easily follow a user and turn accordingly even with sudden direction changes;
- it can adapt its altitude accordingly to the user's height and movements;
- the model is robust enough to other people presence in the camera frame.

The weakness of the model is yaw prediction. The model makes more errors with the yaw prediction probably because multiple users in different recordings face forward while being at the center of the camera frame. This induce a bias in the dataset. This hypothesis is confirmed by real-robot test: the drone ignores user's heading when they are at the center of the camera frame and not moving, while predicts correctly user's heading if they are moving and not in the central section of the camera frame.

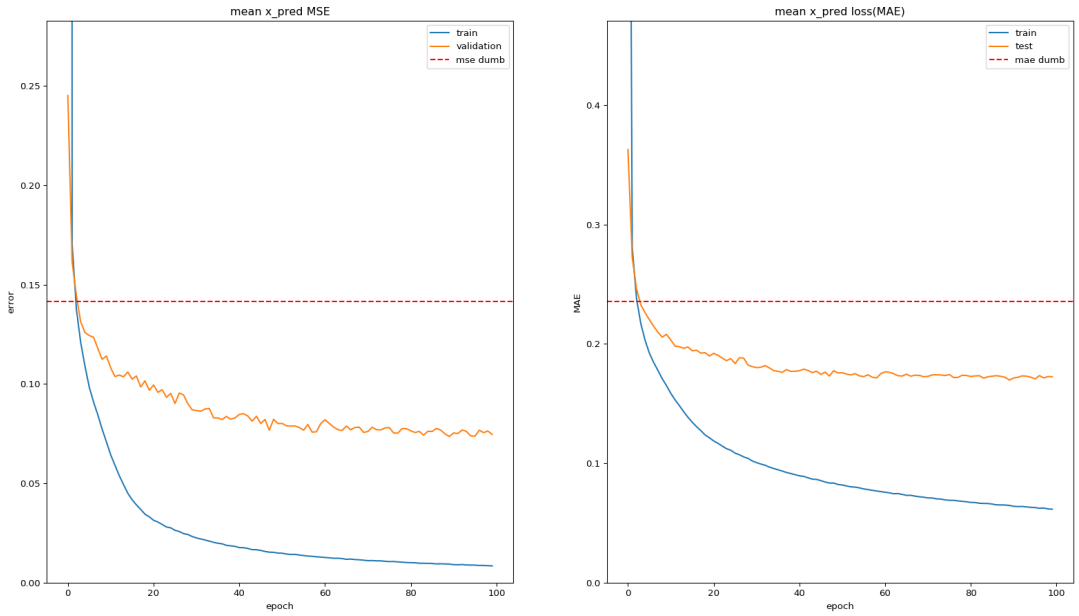
6.2 Future work

To solve this problems we suggest an expansion of the dataset with an increased variety of situations in order to expand the model knowledge. Data could be recorded with a drone flying in a fixed position with a tracked user moving randomly in front of him. In order to improve the real life performance we would need to fine tune the controller parameters to compensate for the physical dynamics of the drone flight or even expand and empower the model to directly control the drone by predicting linear and angular velocities.

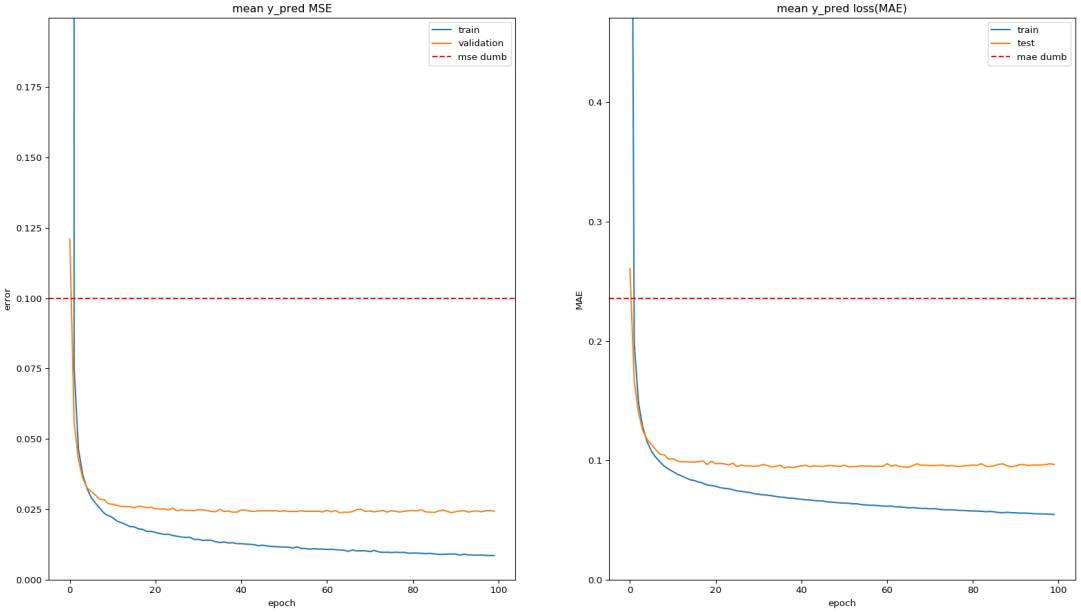
Appendices

Appendix A

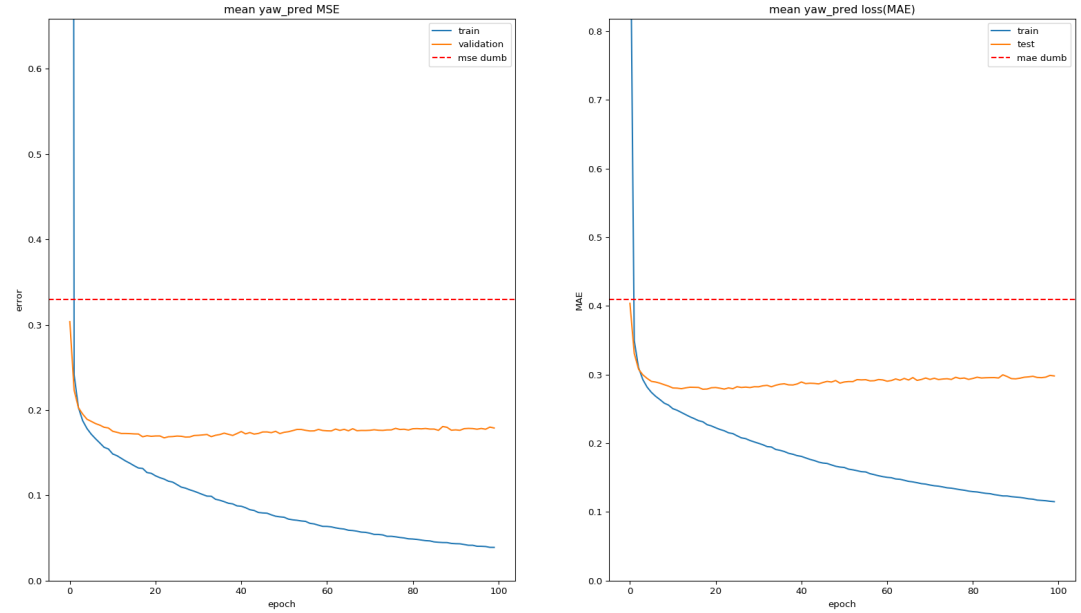
Extra Figures



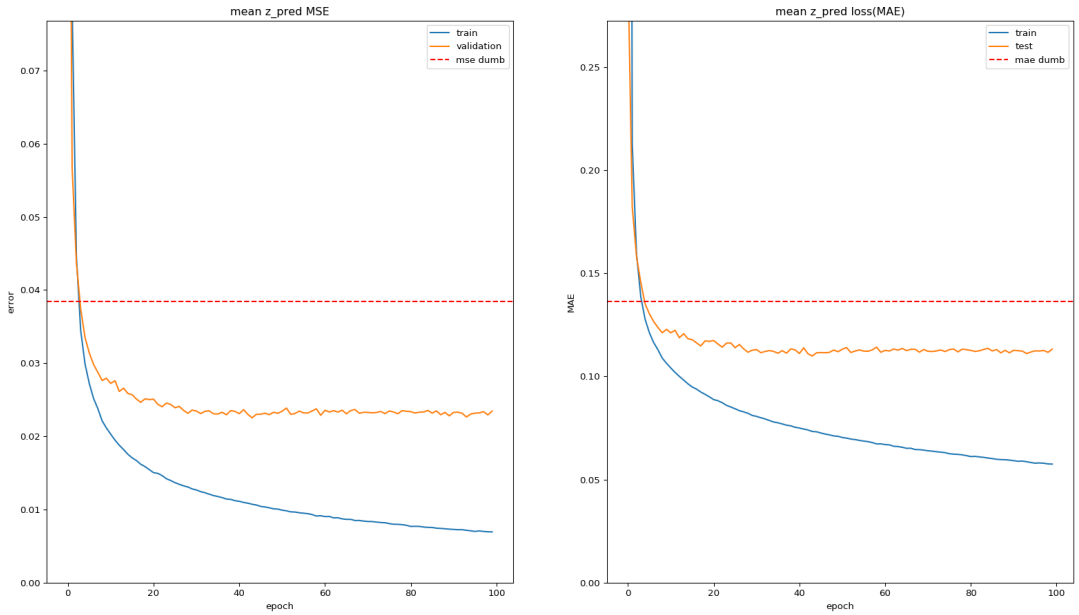
(a) x MSE and MAE plots over 100 epochs



(b) *y* MSE and MAE plots over 100 epochs

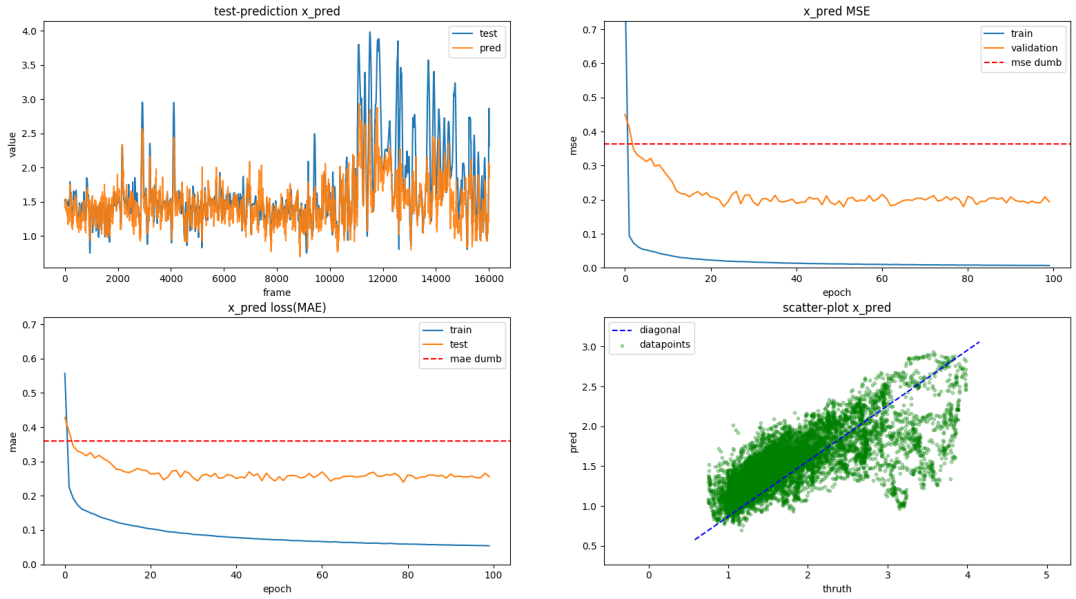


(c) *yaw* MSE and MAE plots over 100 epochs

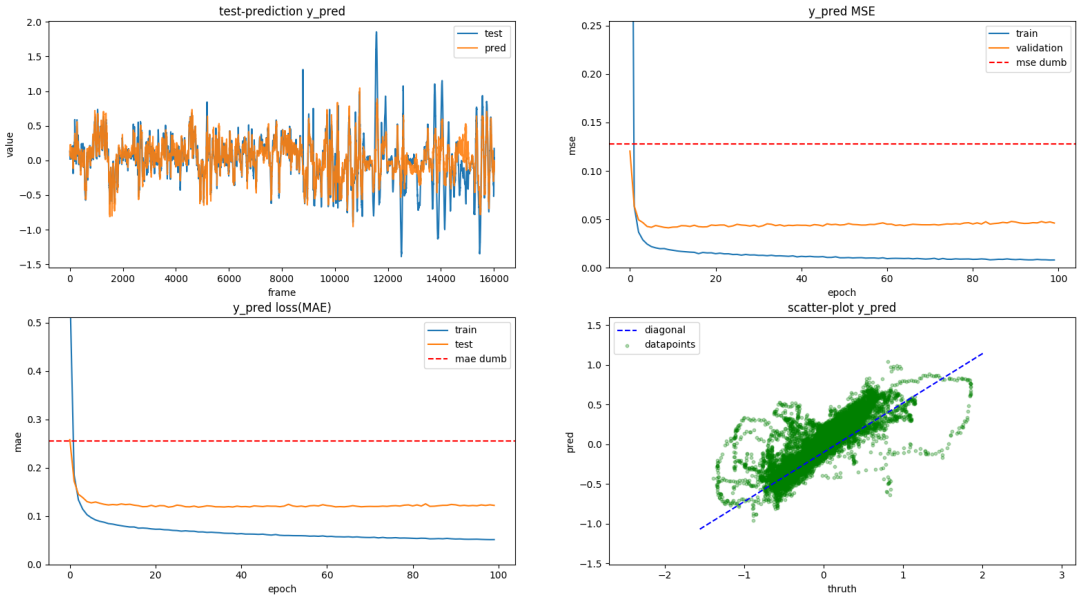


(d) z MSE and MAE plots over 100 epochs

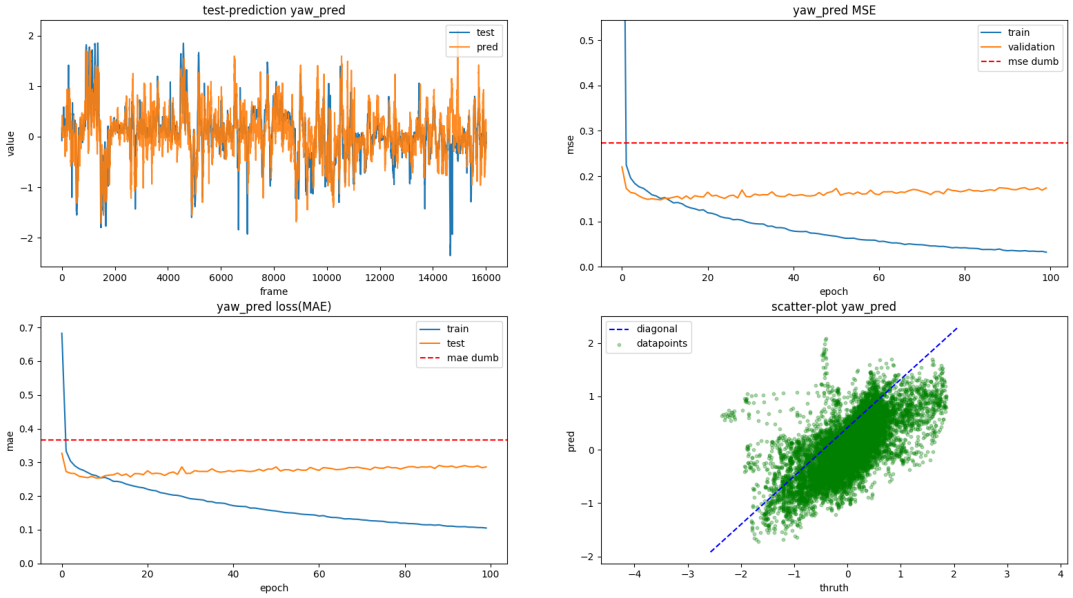
Figure A.1: Graphs of the average loss and MSE for the different target variables. In each graph lines correspond to a specific metric for a specific variable. The line's color indicate the set predicted blue = train, orange = validation, red dashe = dummy regressor on validation.


test-prediction x_pred
test
pred
value
frame
x_pred MSE
train
validation
mse dumb
mse
epoch
x_pred loss(MAE)
train
test
mae dumb
mae
epoch
scatter-plot x_pred
diagonal
datapoints
pred
thruth

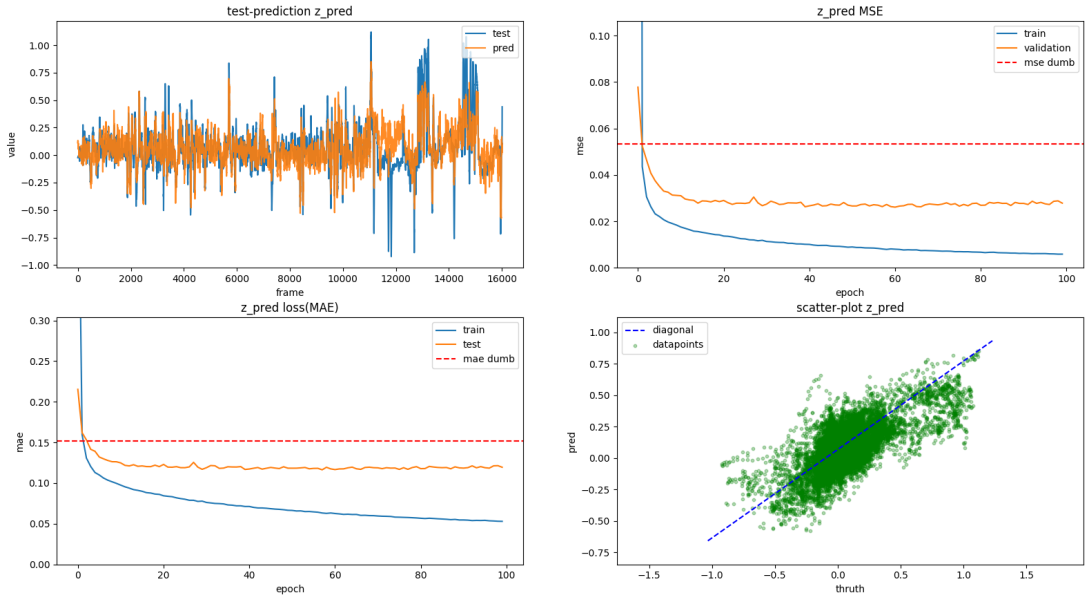
(a) x


test-prediction y_pred
test
pred
value
frame
y_pred MSE
train
validation
mse dumb
mse
epoch
y_pred loss(MAE)
train
test
mae dumb
mae
epoch
scatter-plot y_pred
diagonal
datapoints
pred
thruth

(b) y



(c) *yaw*



(d) *z*

Figure A.2: A set of figures with 4 plots of a fold one figure for each variable. These graphs are described in Section 5.1.1

Appendix B

Other Listings

Other listings excluded from the text for better readability.

`find_nearest` method

Listing B.1: Find Nearest function

```
def find_nearest(array, value):
    """
        find nearest value in array
    Args:
        array: array of values
    value: reference value

    Returns:
        min index of nearest array's element to value
    """
    return (np.abs(array - value)).argmin()
```

`change_frame_reference` method

Listing B.2: Change frame of reference

```
def change_frame_reference(pose_bebop, pose_head):
    """
        Change frame of reference of pose head from World to
           bebop.

    Args:
        pose_bebop: pose of the bebop
        pose_head: pose of the head

    Returns:
        the new pose for head:
            bebop
                T
                head
    """
    position_bebop = pose_bebop[['b_pos_x', 'b_pos_y', 'b_pos_z
       ']].values
```

```
16      quaternion_bebop = pose_bebop[['b_rot_w', 'b_rot_x', '
           b_rot_y', 'b_rot_z']].values
17      position_head = pose_head[['h_pos_x', 'h_pos_y', 'h_pos_z
           ']].values
18      quaternion_head = pose_head[['h_rot_w', 'h_rot_x', 'h_rot_y
           ', 'h_rot_z']].values
19      w_t_b = rospose2homogmat(position_bebop, quaternion_bebop)
20      w_t_h = rospose2homogmat(position_head, quaternion_head)
21      b_t_w = np.linalg.inv(w_t_b)
22      b_t_h = np.matmul(b_t_w, w_t_h)
23
24      return b_t_h
```

VideoCreator Class

Listing B.3: Video creator class

```
1  class VideoCreator:
2      def __init__(self, b_orientation, distances, b_position,
           frame_list, h_orientation, h_position, delta_z, f, title
           ="test.avi"):
3          """
4              Initializer for the class
5          Args:
6              distances: list of distance user-drone
7              b_orientation: bebop orientation array
8              b_position: bebop position array
9              frame_list: camera frame list
10             h_orientation: head orientation array
11             h_position: head orientation array
12             delta_z: height difference list
13             f: bag file name
14             title: video file name
15         """
16         self.fps = 30
17         self.f = f
18         self.video_writer = cv2.VideoWriter(title, cv2.
              VideoWriter_fourcc(*'XVID'), self.fps, (640, 480))
19         self.b_orientation = b_orientation
20         self.b_position = b_position
21         self.frame_list = frame_list
22         self.h_orientation = h_orientation
23         self.h_position = h_position
24         self.distances = distances
25         self.delta_z = delta_z
26
27     def plotting_function(self, i):
28         """
29             Given an index compose the frame for the video.
30         Args:
31             i: frame number
32         """
```

```
33          fig = plt.figure()
34          fig.suptitle("Frame: " + str(i), fontsize=12)
35          axll = fig.add_subplot(2, 2, 1)
36          axl = fig.add_subplot(2, 2, 2)
37          axc = fig.add_subplot(2, 2, 3)
38          axr = fig.add_subplot(2, 2, 4)
39          canvas = FigureCanvas(fig)
40
41          # Central image: here we add the camera feed to the
               video
42          img = Image.open(io.BytesIO(self.frame_list[i]))
43          raw_frame = list(img.getdata())
44          frame = []
45          for b in raw_frame:
46          frame.append(b)
47          reshaped_fr = np.reshape(np.array(frame, dtype=np.int64
               ), (480, 856, 3))
48          reshaped_fr = reshaped_fr.astype(np.uint8)
49          axc.imshow(reshaped_fr)
50          axc.set_axis_off()
51
52          # RIGHT PLOT: here we create the right plot that
               represent the position and heading of the bebop and
               head
53          axr.axis([-2.4, 2.4, -2.4, 2.4], 'equals')
54          h_theta = quat_to_eul(self.h_orientation[i])[2]
55          b_theta = quat_to_eul(self.b_orientation[i])[2]
56          arrow_length = 0.3
57          spacing = 1.2
58          minor_locator = MultipleLocator(spacing)
59          # Set minor tick locations.
60          axr.yaxis.set_minor_locator(minor_locator)
61          axr.xaxis.set_minor_locator(minor_locator)
62          # Set grid to use minor tick locations.
63          axr.grid(which='minor')
64          axr.plot(self.b_position[i].x, self.b_position[i].y, "
               ro", self.h_position[i].x, self.h_position[i].y, "go
               ")
65          axr.arrow(self.h_position[i].x, self.h_position[i].y,
               arrow_length * np.cos(h_theta), arrow_length * np.sin
               (h_theta), head_width=0.05, head_length=0.1, fc='g',
               ec='g')
66          axr.arrow(self.b_position[i].x, self.b_position[i].y,
               arrow_length * np.cos(b_theta), arrow_length * np.sin
               (b_theta), head_width=0.05, head_length=0.1, fc='r',
               ec='r')
67
68          # LEFT PLOT: here we represent the distance on the y
               axis and the heading correction for the drone in
               degrees on the x-axis
69          r_t_h = matrix_method(self.b_position[i], self.
```

```
            b_orientation[i], self.h_position[i], self.
            h_orientation[i])
        horizontal_angle = -math.degrees(math.atan2(r_t_h[1,
            3], r_t_h[0, 3]))

        value_angle_axis = 45
        axl.set_xlim(-value_angle_axis, value_angle_axis)
        axl.set_ylim(0.1, 3)
        axl.set_xlabel('Angle y')
        axl.set_ylabel('Distance')
        axl.plot(horizontal_angle, self.distances[i], 'go')

        axll.set_xlim(-value_angle_axis, value_angle_axis)
        axll.set_ylim(-1, 1)
        axll.set_xlabel('Angle y')
        axll.set_ylabel('Delta z')
        axll.plot(horizontal_angle, self.delta_z[i], 'go')
        # Drawing the plot
        canvas.draw()

        # some additional informations as arrows
        width, height = (fig.get_size_inches() * fig.get_dpi())
            .astype(dtype='int32')
        img = np.fromstring(canvas.tostring_rgb(), dtype='uint8
            ').reshape(height, width, 3)
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        pt1 = (275, 40)
        pt2 = (375, 40)
        if horizontal_angle >= 0:
            cv2.arrowedLine(img, pt1, pt2, (0, 0, 255), 3)
        else:
            cv2.arrowedLine(img, pt2, pt1, (0, 0, 255), 3)

        pt3 = (25, 175)
        pt4 = (25, 225)
        if self.distances[i] < 1.5:
            cv2.arrowedLine(img, pt3, pt4, (0, 255, 0), 3)
        else:
            cv2.arrowedLine(img, pt4, pt3, (0, 255, 0), 3)

        self.video_writer.write(img)
        plt.close(fig)

    def video_plot_creator(self):
        """
            calls frame composers for every frame
            complete video creation
        """
        max_ = len(self.frame_list)
        for i in tqdm.tqdm(range(0, max_)):
            self.plotting_function(i)
```

```
116         self.video_writer.release()
117         cv2.destroyAllWindows()
```

keras_train Script

Listing B.4: keras_train Script

```
1  import math
2  import os
3
4  import numpy as np
5  import pandas as pd
6
7  from tool_to_plot_data import KerasVideoCreator, plot_results
8  from dumb_regressor import dumb_regressor_result
9  from model_creator import model_creator, generator
10 from global_parameters import *
11 from utils import isdebugging
12
13
14 # Cnn method contains the definition, training, testing and
       plotting of the CNN model and dataset
15 def CNNMethod(batch_size, epochs, model_name, save_dir, x_test,
       x_train, y_test, y_train):
16     """
17          Cnn method runs:
18             -train
19             -test
20             -save model as a h5py file
21
22     Args:
23         batch_size: size of a batch
24         epochs: number of epochs
25         model_name: name of the model, used for naming saved
               models
26         save_dir: directory of the running test folder
27         x_test: validation samples
28         x_train: training samples
29         y_test: validation target
30         y_train: training target
31
32     Returns:
33         history: metric history
34         y_pred: prediction on test set
35     """
36     model, _, _ = model_creator(show_summary=True)
37     batch_per_epoch = math.ceil(x_train.shape[0] / batch_size)
38     gen = generator(x_train, y_train, batch_size)
39
40     history = model.fit_generator(generator=gen,
41                                   validation_data=(x_test, [
                                         y_test[:, 0], y_test[:, 1],
```

```
                                        y_test[:, 2], y_test[:,
                                       3]]),
42                                  epochs=epochs,
43                                  steps_per_epoch=
                                       batch_per_epoch)
44      if not os.path.isdir(save_dir):
45          os.makedirs(save_dir)
46      model_path = os.path.join(save_dir, model_name)
47      model.save(model_path)
48      print('Saved trained model at %s ' % model_path)
49
50      scores = model.evaluate(x_test, [y_test[:, 0], y_test[:,
            1], y_test[:, 2], y_test[:, 3]], verbose=1)
51
52      y_pred = model.predict(x_test)
53      print('Test loss:', scores[0])
54      print('Test mse:', scores[1])
55      return history, y_pred
56
57
58  # ------------------- Main ----------------------
59  def main():
60      """
61          -read pickle file for train and validation
62          -calls method for train and predict
63          -calls method to run dumb prediction
64          -calls method to create a video for qualitative
                evaluation
65          -calls method to plot data for quantitative evaluation
66      """
67      train = pd.read_pickle("./dataset/train.pickle").values
68      validation = pd.read_pickle("./dataset/validation.pickle").
            values
69
70      if isdebugging():
71          print("debugging-settings")
72          batch_size = 128
73          epochs = 2
74      else:
75          batch_size = 64
76          epochs = 100
77      save_dir = os.path.join(os.getcwd(), 'saved_models')
78      model_name = 'keras_bebop_trained_model.h5'
79      # split between train and test sets:
80      x_train = 255 - train[:, 0]  # otherwise is inverted
81      x_train = np.vstack(x_train[:]).astype(np.float32)
82      x_train = np.reshape(x_train, (-1, image_height,
            image_width, 3))
83      y_train = train[:, 1]
84      y_train = np.asarray([np.asarray(sublist) for sublist in
            y_train])
```

```
85
86      x_test = 255 - validation[:, 0]
87      x_test = np.vstack(x_test[:]).astype(np.float32)
88      x_test = np.reshape(x_test, (-1, image_height, image_width,
            3))
89      y_test = validation[:, 1]
90      y_test = np.asarray([np.asarray(sublist) for sublist in
            y_test])
91
92      print('x_train shape: ' + str(x_train.shape))
93      print('train samples: ' + str(x_train.shape[0]))
94      print('test samples:  ' + str(x_test.shape[0]))
95
96      history, y_pred = CNNMethod(batch_size, epochs, model_name,
            save_dir, x_test, x_train, y_test, y_train)
97      dumb_metrics = dumb_regressor_result(x_test, x_train,
            y_test, y_train)
98      vidcr_test = KerasVideoCreator(x_test=x_test, targets=
            y_test, preds=y_pred, title="./video/test_result.avi")
99      vidcr_test.video_plot_creator()
100     plot_results(history, y_pred, y_test, dumb_metrics)
101
102
103 if __name__ == "__main__":
104     main()
```

keras_crossvalidation Script

Listing B.5: keras_crossvalidation Script

```
1  import math
2  import os
3  from datetime import datetime
4
5  import numpy as np
6  import pandas as pd
7  from global_parameters import *
8
9  from dumb_regressor import dumb_regressor_result
10 from model_creator import model_creator, generator
11 from tool_to_plot_data import history_data_plot_crossvalidation
       , plot_results_cross, KerasVideoCreator
12
13
14 def CNNMethod(batch_size, epochs, model_name, save_dir, x_test,
       x_train, y_test, y_train, i):
15     """
16         Cnn method runs a fold of the k-fold crossvalidation:
17             -train
18             -test
19             -save model as a h5py file
20             -save hyperparameters values as txt file
```

```
21              -calls method to run dumb prediction
22              -calls method to create a video for qualitative
                   evaluation
23              -calls method to plot data for quantitative
                   evaluation
24
25      Args:
26          batch_size: size of a batch
27          epochs: number of epochs
28          model_name: name of the model, used for naming saved
               models
29          save_dir: directory of the running test folder
30          x_test: validation samples
31          x_train: training samples
32          y_test: validation target
33          y_train: training target
34          i: index of the i-th fold of cross validation
35
36      Returns:
37          history.history: history of metrics of i-th fold run
38          dumb_metrics: list of metrics results after dumb
               regression
39      """
40      print("k-fold:" + str(i))
41      model, lr, _ = model_creator(show_summary=True)
42      if i == 0:
43          # plot_model(model.layers[1], to_file=save_dir + '/
               model_seq.png')
44          # plot_model(model, to_file=save_dir + '/model_out.png
               ')
45          with open(save_dir + "/model_info.txt", "w+") as
               outfile:
46              outfile.write("Hyperparameters\n")
47              outfile.write("== == == == == == == == == == ==\
                   n")
48              outfile.write("learning_rate:" + str(lr) + "\n")
49              outfile.write("batch size:" + str(batch_size) + "\n
                   ")
50              outfile.write("epochs:" + str(epochs) + "\n")
51              outfile.write("== == == == == == == == == == ==\
                   n")
52              model.layers[1].summary(print_fn=lambda x: outfile.
                   write(x + '\n'))
53              model.summary(print_fn=lambda x: outfile.write(x +
                   '\n'))
54              outfile.close()
55      batch_per_epoch = math.ceil(x_train.shape[0] / batch_size)
56      gen = generator(x_train, y_train, batch_size)
57      history = model.fit_generator(generator=gen,
58                                    validation_data=(x_test, [
                                       y_test[:, 0], y_test[:, 1],
```

```
                                        y_test[:, 2], y_test[:,
                                       3]]),
59                                  epochs=epochs,
60                                  steps_per_epoch=
                                       batch_per_epoch)
61
62      # Save model and weights
63      if not os.path.isdir(save_dir):
64          os.makedirs(save_dir)
65      model_path = os.path.join(save_dir, model_name)
66      model.save(model_path)
67      print('Saved trained model at %s ' % model_path)
68
69      # Score trained model.
70      scores = model.evaluate(x_test, [y_test[:, 0], y_test[:,
            1], y_test[:, 2], y_test[:, 3]], verbose=1)
71      y_pred = model.predict(x_test)
72      print('Test loss:', scores[0])
73      print('Test mse:', scores[1])
74
75      vidcr_test = KerasVideoCreator(x_test=x_test, targets=
            y_test, preds=y_pred, title=save_dir + "/result_model_" +
            str(i) + "/test_result.avi")
76      vidcr_test.video_plot_creator()
77      dumb_metrics = dumb_regressor_result(x_test, x_train,
            y_test, y_train)
78      plot_results_cross(history, y_pred, y_test, dumb_metrics,
            save_dir, i)
79      return history.history, dumb_metrics
80
81
82  def crossValidation(k_fold, batch_size, epochs):
83      """
84          Runs all folds of a k-fold crossvalidation
85          -creates save directory of the run usign datetime as
                foldername
86          -for each fold:
87              -reads the .pickle files and compose training and
                    validation sets
88              -calls CNNMethod
89              -saves running time of fold in computation_time.txt
90          -saves history data from all runs
91          -plot crossvalidation mean metrics results
92          -save crossvalidation time in computation_time.txt
93
94      Args:
95          k_fold: number of folds
96          batch_size: size of a batch
97          epochs: number of epochs
98
99      Returns:
```

```
100         nothing
101     """
102     start_time = datetime.now()
103     save_path = 'saves/' + datetime.now().strftime("%Y-%m-%d-%H
            -%M-%S")
104     try:
105         os.makedirs(save_path)
106     except OSError:
107         if not os.path.isdir(save_path):
108             raise
109     for i in range(k_fold):
110         try:
111             os.makedirs(save_path + "/result_model_" + str(i))
112         except OSError:
113             if not os.path.isdir(save_path + "/result_model_" +
                    str(i)):
114                 raise
115     path = "./dataset/crossvalidation/"
116     files = [f for f in os.listdir(path) if f[-7:] == '.pickle
            ']
117     dumb_list = []
118     history_list = []
119     save_dir = os.path.join(os.getcwd(), save_path)
120     if not files:
121         print('No bag files found!')
122         return None
123     for i in range(k_fold):  # test selection,
124         e_start_time = datetime.now()
125         x_test_list = []
126         x_train_list = []
127 
128         # create test and train set
129         for f in files:  # train selection
130             section = pickle_sections[f[:-7]]
131 
132             if section == i:
133                 x_test_list.append(pd.read_pickle("./dataset/
                        crossvalidation/" + f))
134 
135             else:
136                 x_train_list.append(pd.read_pickle("./dataset/
                        crossvalidation/" + f))
137         train = pd.concat(x_train_list).values
138         validation = pd.concat(x_test_list).values
139 
140         model_name = 'keras_bebop_trained_model_' + str(i) + '.
                h5'
141         x_train = 255 - train[:, 0]  # otherwise is inverted
142         x_train = np.vstack(x_train[:]).astype(np.float32)
143         x_train = np.reshape(x_train, (-1, image_height,
                image_width, 3))
```

```
144         y_train = train[:, 1]
145         y_train = np.asarray([np.asarray(sublist) for sublist
                in y_train])
146         x_test = 255 - validation[:, 0]
147         x_test = np.vstack(x_test[:]).astype(np.float32)
148         x_test = np.reshape(x_test, (-1, image_height,
                image_width, 3))
149         y_test = validation[:, 1]
150         y_test = np.asarray([np.asarray(sublist) for sublist in
                y_test])
151         print('x_train shape: ' + str(x_train.shape))
152         print('train samples: ' + str(x_train.shape[0]))
153         print('test samples:  ' + str(x_test.shape[0]))
154         history, dumb_results = CNNMethod(batch_size, epochs,
                model_name, save_dir, x_test, x_train, y_test,
                y_train, i)
155         history_list.append(history)
156         dumb_list.append(dumb_results)
157         e_end_time = datetime.now()
158         print("")
159         print("")
160         print("----k-Fold " + str(i) + " info:")
161         print("started: " + e_start_time.strftime("%Y-%m-%d-%H
                -%M-%S"))
162         print("ended:   " + e_end_time.strftime("%Y-%m-%d-%H-%M
                -%S"))
163         h = divmod((e_end_time - e_start_time).total_seconds(),
                3600)  # hours
164         m = divmod(h[1], 60)  # minutes
165         s = m[1]  # seconds
166         print("lasted:  %d hours, %d minutes, %d seconds" % (h
                [0], m[0], s))
167         if i < 4:
168             eta_h = divmod((k_fold - i - 1) * (e_end_time -
                    e_start_time).total_seconds(), 3600)  # hours
169             eta_m = divmod(eta_h[1], 60)  # minutes
170             eta_s = eta_m[1]  # seconds
171             print("ETA:     %d hours, %d minutes, %d seconds" %
                    (eta_h[0], eta_m[0], eta_s))
172         print("----")
173         print("")
174         print("")
175         with open(save_dir + "/computation_time.txt", "w+") as
                outfile:
176             outfile.write("----k-Fold " + str(i) + " info:")
177             outfile.write("started: " + e_start_time.strftime
                    ("%Y-%m-%d-%H-%M-%S"))
178             outfile.write("ended:   " + e_end_time.strftime("%Y
                    -%m-%d-%H-%M-%S"))
179             outfile.write("lasted:  %d hours, %d minutes, %d
                    seconds" % (h[0], m[0], s))
```

```
180             outfile.write("----")
181             outfile.close()
182     hist_df = pd.DataFrame(history_list)
183     pickle_path = save_dir + "/metrics_history.pickle"
184     hist_df.to_pickle(pickle_path)
185     print("history saved: " + pickle_path)
186     history_data_plot_crossvalidation(history_list, dumb_list,
            save_dir)
187     end_time = datetime.now()
188     print("")
189     print("")
190     print("---- total computation time:")
191     print("started: " + start_time.strftime("%Y-%m-%d-%H-%M-%S
            "))
192     print("ended:   " + end_time.strftime("%Y-%m-%d-%H-%M-%S"))
193     h = divmod((end_time - start_time).total_seconds(), 3600)
            # hours
194     m = divmod(h[1], 60)  # minutes
195     s = m[1]  # seconds
196     print("lasted:  %d hours, %d minutes, %d seconds" % (h[0],
            m[0], s))
197     print("----")
198     with open(save_dir + "/computation_time.txt", "w+") as
            outfile:
199         outfile.write("---- total computation time:")
200         outfile.write("started: " + start_time.strftime("%Y-%m
                -%d-%H-%M-%S"))
201         outfile.write("ended:   " + end_time.strftime("%Y-%m-%d
                -%H-%M-%S"))
202         outfile.write("lasted:  %d hours, %d minutes, %d
                seconds" % (h[0], m[0], s))
203         outfile.write("----")
204         outfile.close()
205
206
207 # ------------------- Main ----------------------
208 def main():
209     """
210     -Setup k-fold cross validation parameters
211     -calls crossValidation()
212     """
213     k_fold = 5
214     batch_size = 64
215     num_classes = 4
216     epochs = 100
217     # epochs = 2
218     crossValidation(k_fold, batch_size, num_classes, epochs)
219
220
221 if __name__ == "__main__":
222     main()
```

model_creator Script

Listing B.6: model_creator Script

```
import keras

from keras.models import Sequential, Model
from keras.layers import *
from global_parameters import *


def model_creator(show_summary=False, old=False):
    """
        Generate the model.
        Available two architcture
            -old
            -new
    Args:
        show_summary: if true, show keras model summary in console
        old: if true create old architecture

    Returns:

    """
    if old:
        seq_model = create_sequential()
        model_input = Input((image_height, image_width, 3))
        out_sequential = seq_model(model_input)
        y_1 = (Dense(1, activation='linear', name="distance_pred"))(out_sequential)
        y_2 = (Dense(1, activation='linear', name="angle_pred"))(out_sequential)
        y_3 = (Dense(1, activation='linear', name="height_pred"))(out_sequential)
        model = Model(inputs=model_input, outputs=[y_1, y_2, y_3])
        learn_rate = 0.001
        decay = 1e-6
        opt = keras.optimizers.rmsprop(lr=learn_rate, decay=decay)
        model.compile(loss='mean_absolute_error',
                      optimizer=opt,
                      metrics=['mse'])
        if show_summary:
            model.summary()
    else:  # NEW
        seq_model = create_sequential()
        model_input = Input((image_height, image_width, 3))
        out_sequential = seq_model(model_input)
        y_1 = (Dense(1, activation='linear', name="x_pred"))(out_sequential)
```

```
42          y_2 = (Dense(1, activation='linear', name="y_pred"))(
                out_sequential)
43          y_3 = (Dense(1, activation='linear', name="z_pred"))(
                out_sequential)
44          y_4 = (Dense(1, activation='linear', name="yaw_pred"))(
                out_sequential)
45          model = Model(inputs=model_input, outputs=[y_1, y_2,
                y_3, y_4])
46          learn_rate = 0.00005
47          opt = keras.optimizers.Adam(lr=learn_rate)
48          model.compile(loss='mean_absolute_error',
49                        optimizer=opt,
50                        metrics=['mse'])
51          if show_summary:
52              model.summary()
53
54      return model, learn_rate, 0
55
56
57  def create_sequential():
58      """
59          Create sequential part of the model architecture
60      Returns:
61          model: keras Sequential object
62      """
63      model = Sequential()
64      model.add(Conv2D(10, (6, 6), padding='same', input_shape=(
            image_height, image_width, 3), name="1_conv"))
65      model.add(Activation('relu'))
66      model.add(MaxPooling2D(pool_size=(3, 3), name="1_pool"))
67      model.add(Conv2D(15, (6, 6), padding='same', name="2_conv")
            )
68      model.add(Activation('relu'))
69      model.add(MaxPooling2D(pool_size=(2, 2), name="2_pool"))
70      model.add(Conv2D(20, (6, 6), padding='same', name="3_conv")
            )
71      model.add(Activation('relu'))
72      model.add(MaxPooling2D(pool_size=(2, 2), name="3_pool"))
73      model.add(Flatten())
74      model.add(Dense(256, name="1_dense"))
75      model.add(Activation('relu'))
76      model.add(Dense(64, name="2_dense"))
77      model.add(Activation('relu'))
78      return model
79
80
81  def data_augmentor(frame, target):
82      """
83          recieves frame and targets with p=50% flip vertically
84      Args:
85          frame: image
```

```
86          target:  target for CNN
87
88      Returns:
89          frame and targers eventually flipped
90      """
91      if np.random.choice([True, False]):
92          frame = np.fliplr(frame)  # IMG
93          target[1] = -target[1]  # Y
94          target[3] = -target[3]  # Relative YAW
95      return frame, target
96
97
98  def generator(samples, targets, batch_size, old=False):
99      """
100         Genereator of minibatches of size batch_size
101     Args:
102         samples: sample array
103         targets: targets array
104         batch_size: batch size
105         old: if true genereate data for old architecture
106     Yields:
107         batch of samples and array of batch of targets
108     """
109     if old:  # OLD
110         while True:
111             indexes = np.random.choice(np.arange(0, samples.
                    shape[0]), batch_size)
112             batch_samples = samples[indexes]
113             batch_targets = targets[indexes]
114             for i in range(0, batch_samples.shape[0]):
115                 batch_samples[i], batch_targets[i] =
                        data_augmentor(batch_samples[i],
                        batch_targets[i])
116             yield batch_samples, [batch_targets[:, 0],
                    batch_targets[:, 1], batch_targets[:, 2]]
117     else:  # NEW
118         while True:
119             indexes = np.random.choice(np.arange(0, samples.
                    shape[0]), batch_size)
120             batch_samples = samples[indexes]
121             batch_targets = targets[indexes]
122             for i in range(0, batch_samples.shape[0]):
123                 batch_samples[i], batch_targets[i] =
                        data_augmentor(batch_samples[i],
                        batch_targets[i])
124             yield batch_samples, [batch_targets[:, 0],
                    batch_targets[:, 1], batch_targets[:, 2],
                    batch_targets[:, 3]]
```

bag_to_pickle Method

Listing B.7: Extract that shows the method bag_to_pickle

```
def bag_to_pickle(f):
"""
Core method used to transforms and saves a bag file into a .
   pickle dataset file
Args:
f: file name e.g. "7.bag"
"""
path = bag_file_path[f[:-4]]
print("\nreading bag: " + str(f))
datacr = DatasetCreator()
with rosbag.Bag(path + f) as bag:
bag_df_dict = get_bag_data_pandas(bag)
data_vec = processing(bag_df_dict=bag_df_dict, data_id=f[:-4],
   f=f)
datacr.generate_data(data_vec=data_vec)
datacr.save_dataset(flag_train="cross", title=f[:-4] + ".pickle
   ")

print("\nCompleted pickle " + str(f[:-4]))
```

processing Method

Listing B.8: processing method

```
def processing(bag_df_dict, data_id, f):
"""
Process data from dictionary bag_df_dict into a data vector
Args:
bag_df_dict: dictionary of Pandas dataframes
data_id: id of the bag file processed
f: bag file name, used as key for dictionary
Returns:
data vector: vector of tutples (image, (targe_x, target_y,
   target_z, target_relative_yaw)
"""
camera_t = bag_df_dict["camera_df"].index.values
bebop_t = bag_df_dict["bebop_df"].index.values
head_t = bag_df_dict["head_df"].index.values
data_vec = []
max_ = bag_end_cut[f[:-4]]
min_ = bag_start_cut[f[:-4]]
for i in tqdm.tqdm(range(min_, max_), desc="processing data " +
    str(data_id)):
b_id = find_nearest(bebop_t, camera_t[i])
h_id = find_nearest(head_t, camera_t[i])
head_pose = bag_df_dict["head_df"].iloc[h_id]
bebop_pose = bag_df_dict["bebop_df"].iloc[b_id]
img = bag_df_dict["camera_df"].iloc[i].values[0]
b_t_h = change_frame_reference(bebop_pose, head_pose)
quaternion_bebop = bebop_pose[['b_rot_x', 'b_rot_y', 'b_rot_z',
    'b_rot_w']].values
```

```
quaternion_head = head_pose[['h_rot_x', 'h_rot_y', 'h_rot_z', '
    h_rot_w']].values
_, _, head_yaw = quat_to_eul(quaternion_head)
_, _, bebop_yaw = quat_to_eul(quaternion_bebop)
relative_yaw = (head_yaw - bebop_yaw - np.pi)
if relative_yaw < -np.pi:
relative_yaw += 2 * np.pi
target_position = b_t_h[:-1, -1:].T[0]
target = (target_position[0], target_position[1],
    target_position[2], relative_yaw)
data_vec.append((img, target))
return data_vec
```

dumb_regressor Script

Listing B.9: dumb_regressor Script

```
from sklearn.dummy import DummyRegressor
from sklearn.metrics import mean_absolute_error,
    mean_squared_error
import numpy as np


def dumb_regressor_result(x_test, x_train, y_test, y_train):
    """
          Dumb regressor, predict only the mean value for each
             target variable,
          returns MAE and MSE metrics per each variable.

          Args:
            x_test: validation samples
            x_train: training samples
            y_test: validation target
            y_train: training target

          Returns:
            dumb_metrics: list of metrics results after dumb
               regression
    """
    dumb_reg = DummyRegressor()
    fake_data = np.zeros((x_train.shape[0], 1))
    fake_test = np.zeros((1, 1))
    dumb_reg.fit(fake_data, y_train)
    dumb_pred = dumb_reg.predict(fake_test)[0]
    dumb_pred_x = np.full((x_test.shape[0], 1), dumb_pred[0])
    dumb_pred_y = np.full((x_test.shape[0], 1), dumb_pred[1])
    dumb_pred_z = np.full((x_test.shape[0], 1), dumb_pred[2])
    dumb_pred_yaw = np.full((x_test.shape[0], 1), dumb_pred[3])
    dumb_mse_x = mean_squared_error(y_test[:, 0], dumb_pred_x)
    dumb_mae_x = mean_absolute_error(y_test[:, 0], dumb_pred_x)
    dumb_mse_y = mean_squared_error(y_test[:, 1], dumb_pred_y)
    dumb_mae_y = mean_absolute_error(y_test[:, 1], dumb_pred_y)
```

```
33      dumb_mse_z = mean_squared_error(y_test[:, 2], dumb_pred_z)
34      dumb_mae_z = mean_absolute_error(y_test[:, 2], dumb_pred_z)
35      dumb_mse_yaw = mean_squared_error(y_test[:, 3],
            dumb_pred_yaw)
36      dumb_mae_yaw = mean_absolute_error(y_test[:, 3],
            dumb_pred_yaw)
37      dumb_metrics = [[dumb_mse_x, dumb_mae_x], [dumb_mse_y,
            dumb_mae_y], [dumb_mse_z, dumb_mae_z], [dumb_mse_yaw,
            dumb_mae_yaw]]
38      return dumb_metrics
```

References

[1] A. Giusti, J. Guzzi, D. C. Ciresan, F. He, J. P. Rodriguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661–667, 2016.

[2] A. Loquercio, A. Maqueda, C. R. Del Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," vol. PP, pp. 1–1, 01 2018.

[3] J. Guzzi, "Drone arena." `https://github.com/jeguzzi/drone_arena`.

[4] S. S. Haykin, *Neural networks and Learning Machines*. Upper Saddle River, NJ: Pearson Education, third ed., 2009.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[6] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *Journal of Physiology (London)*, vol. 195, pp. 215–243, 1968.

[7] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[8] R. Baldwin, "Flying a drone is easier when the battery doesn't die right away." `https://www.engadget.com/2015/11/23/parrot-bebop-2/`, November 2015.

[9] B. Popper, "Parrot bebop 2 review: fun, fine, and fatally flawed." `https://www.theverge.com/2016/1/22/10814282/parrot-bebop-2-drone-review`, January 2016.

[10] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[11] X. L. Amin Jourabloo, "Large-pose face alignment via cnn-based dense 3d model fitting," 2016.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

Acronyms

CNN Convolutional Neural Network III
ROS Robot Operating System III
NN Neural Network 3
FOV Field of View V
MoCap Motion Capture 11
IDSIA Istituto Dalle Molle di Studi sull'Intelligenza Artificiale XI
RNN Recurrent Neural Network 3
OS Operating System 14
WLAN Wireless Local Area Network 9
MIMO Multiple-Input and Multiple-Output 9
GNSS Global Navigation Satellite System 10
GPS Global Positioning system 10
wrt. with respect to 7
IR Infrared 1
ReLU Rectified Linear Unit 5
MSE Mean Squared Error VI
MAE Mean Absolute Error VI
OpenCV Open Source Computer Vision Library IV
ASIC Application Specific Integrated Circuit 39
DoF Degree of Freedom 11
Adam Adaptive Moment Estimation 33
SVM Support Vector Machines 3

Thanks to

After all these pages it's time to thanks all the people that helped me and sustained me during my Master Studies.

First of all thanks to my Family, you've been always there for me, supporting me in any way possible.

Thanks to Tatiana my lovely girlfriend that has always been with me during this difficult journey.

Thanks to IDSIA and in particular Alessandro and Jerome for the opportunity of doing my thesis on something so interesting.

Thanks to Professor Pezzè and double degree coordinators for the unique opportunity that is the double degree program.

Thanks to my best friend (and proof-reader)Andy, to Samu, Sandy, Carola, Fabio, Fede, Mastro and all my friends. Thanks to guys and gals from USI, USI Home and from Bicocca. Thanks to Mirko, we made it.

Thanks to all the others, that for space reason I can't mention, but have been by my side in this years.