# Embedded Implementation of Reactive End-to-End Visual Controller for Nano-Drones

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Visual Computing

presented by
Nicky Zimmerman

under the supervision of
Prof. Alessandro Giusti
co-supervised by
Dr. Jérôme Guzzi

September 2020

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Nicky Zimmerman
Lugano, Yesterday September 2020

*To Green Day, for teaching me how to march to my own beat.*

"Hidden talent counts for nothing"

-Nero

# Abstract

We demonstrate a pipeline for deploying deep learning algorithms to edge devices: in particular, we rework and port an existing algorithm (previously executed off-board), to run on-board a nano-drone equipped with ultra-low-power processor. The algorithm receives input from a forward-facing, low-fidelity camera; its output is used to control the nano-drone to track an operator freely moving in an environment.

To achieve this goal, we first acquire training datasets in a room equipped with a motion-tracking system, by extending an existing data acquisition infrastructure. Using this data, we train a convolutional neural network to predict the user pose relative to the camera, given a video frame. We then deploy the model to the embedded platform: this requires quantizing the model and representing it with hardware-specific C code. We finally integrate the system on the nano-drone, feeding the predictions from the model to a pose-tracking controller.

We evaluate the system with quantitative and qualitative experiments. We also explore factors that impact the quality of deployment to edge devices, such as accuracy, performance and energy efficiency.

# Acknowledgements

My first and foremost gratitude goes to my main advisors, Alessandro and Jerome, for finding the perfect balance between guidance and freedom to explore on my own. In addition to providing a solid technical support, they also handled my emotional meltdowns remarkably well.

Following are the people that inspired this work, my co-advisors, Daniele Palossi and Francesco Conti. While they suffered from my temper to a lesser extent than my advisors, their patience is nonetheless, highly appreciated.

During this process, I was fortunate enough to gain not only knowledge, but also new friendships. I had the pleasure of working with Hanna Mueller for nearly a year, and in addition to being an outstanding engineer, she is also a great friend. Her beloved plush penguin inspired the creation of PenguiNet, and her support throughout my thesis is invaluable. I expect this friendship to outlast our time in academia.

Thanks to Prof. Piotr Didyk for advising on the design of the Gapuino shield and printing the case in his fabrication lab. Thankfully, we only needed two printing attempts to get it right.

Last but not least, my master directors, Prof. Kai Hormann and Prof. Evanthia Papadopoulou. Thank you for being inspirational figures, sympathetic ear and working relentlessly to improve the experience of students in the faculty.

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Usage of nano-drones (Fig. 1.1) is becoming more and more prevalent in both industry and research. The small size of these miniature quadrotors make them perfect candidates for operating in indoor environments, and in close proximity to humans. Due to their modest proportions they are perceived as non-threatening by users, even in close distance. This quality enables the exploration of interesting human-drone interactions that are impossible with standard-sized drones.

Nano-sized drones with autonomous capabilities are rare, but recent advancements made this concept feasible. Independent, self-navigating nano-drones are ideal for many applications. First, replacing costly sensor networks. Instead of having numerous static sensors, a nano-drone can navigate in the area of interest, collect data, and forward it, possibly even filtering it and saving bandwidth and server load. Second, in cluttered, or heavily populated areas, like smart building. Their small size is non-intimidating, and even if collision occurs, it would be harmless. Another use includes deploying nano-drones in search and rescue mission, to collect information through tight passages where humans can not pass (e.g. collapsed buildings). There is also the possibility of integrating nano-drones as part of art and entertainment.

There are ultra low power platforms designed to perform more complex computations on edge devices, complying with the power and size constraints. If we wish to bring the power of artificial intelligence to nano-drones, and enable their autonomy, we also need tailored software. The resource-hungry deep learning frameworks must be highly optimized and customized in order to execute on the resource-starved computational hardware. This process is complicated, and it requires niche, domain-specific knowledge. To make the deployment of deep learning algorithms to nano-drones more accessible, there is a need for a simplified software pipeline, and this is where we step in.

## 1.2 Challenges

Self-navigating standard-size drones are an active area of research. The problem of mapping the environment, localizing the drone and planning a secure trajectory is fundamental in computer vision and robotics. The popular approaches for this complex task are resource-hungry, both

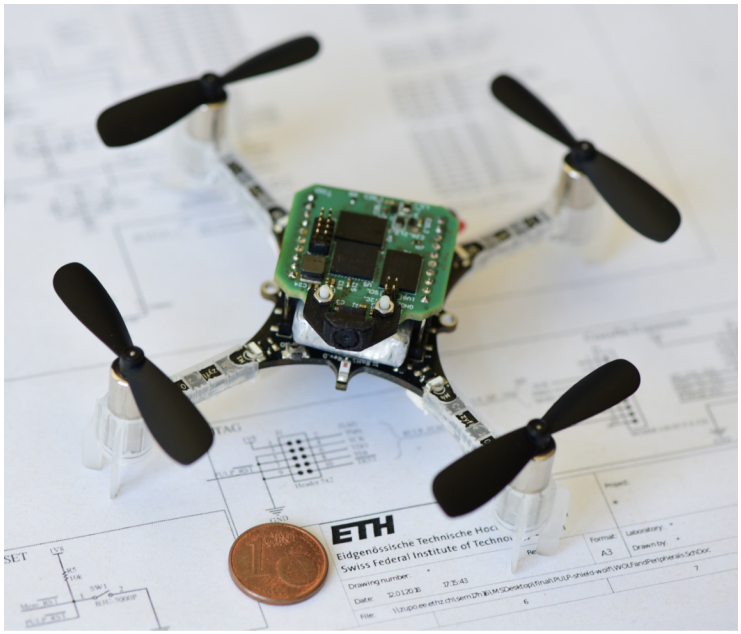


Figure 1.1. Nano-drones by Bitcraze, compared to a 1 cent coin

in terms of memory and compute requirements. This makes the deployment to low-cost edge-nodes impractical, due to the gap between the required resources and the available, on-board resources.

Targeting ultra-low-power platforms presents several challenges. The small payload limits the weight of the battery, compute unit and sensors. The sensors mounted on them are of lesser quality than the high-end sensors used with standard drones. The battery has smaller capacity, introducing the need for clever energy budgeting. The processing unit must be small, energy-efficient and light-weight. These limitations translate to significant reduction in the computational resources, which warrants power-aware, memory-aware algorithms that are capable of achieving fast-responding behavior.

## 1.3 Objective

The goal of this work is to implement and assess a procedure that ports standard PyTorch-modelled convolutional neural networks to ultra-low-power devices, specifically the PULP platforms. This pipeline is meant to simplify and standardize the deployment. It allows practitioners that are not familiar with advanced embedded development techniques to execute deep learning algorithms on nano-drones. The quality of deployment is measured for a human-drone interaction task, where the drone is expected to hover at a comfortable distance from a free moving user, adjusting itself to face the user's head orientation.

The main focus is on showing that full precision PyTorch-based convolutional neural networks can be minimized to fit the computational limitations of edge devices, and re-implemented as memory-optimized c code, while preserving the quality of results.

During the process of research and implementation, we explored factors that can affect the

performance and accuracy of the models, concentrating on aspects that are most relevant to embedded platforms.

We also provide a code base, covering every stage of the deployment process - data collection, data processing, training, validation, visualization, quantization, conversion to c and the on-board application.

## 1.4 Previous Work

### 1.4.1 Vision-based Control of a Quadrotor in User Proximity

In this work [Mantegazza et al., 2019] the authors explore the benefits of end-to-end learning vs. mediated learning, for the task of following a free-moving person. The end-to-end approach learns the correct control from an input image, and the mediated approach learns to estimate the relative pose of the subject and uses the prediction as an input to a dedicated controller. In the paper, it was shown that for that specific task, both mediated and end-to-end approaches performed similarly, with not clear advantage to any approach. Even though the authors deal with a specific reactive control task, it is an interesting, real-world task, that has common attributes with other popular sub-problems in the domain of mobile robotics. The experiment was done using the Parrot Bebop 2 quadrotor, sending the captured images to a server via WiFi. The inference and control was done on the server, and communicated back to the drone through ROS interface. The authors provide the datasets and code in this repository. Since the model was not given an explicit name in the paper, we will refer to it as ProximityNet for readability purposes.

### 1.4.2 DroNet

Autonomous navigation is a well-studied area of research, and the leading approach revolved around localizing the robot, mapping the environment, and planning a secure trajectory. These methods are extremely compute-intense, and impractical for platforms with limited resources. Recent studies have shown that algorithms based on convolutional neural networks are a favorable alternative, as they are less computationally demanding than the localization-mapping-planning approach, and provide basic navigational capabilities for small drones. DroNet [Loquercio et al., 2018] is such visual navigation algorithm. By using data collected by bicycles and cars, this convolutional neural network was trained to fly autonomously through the city streets.

### 1.4.3 PULP-DroNet

Small-sized drones are becoming increasingly popular in industry and research. While previously pocket-sized drones depended on external signal or computation to navigate, the work of PULP-DroNet [Palossi et al., 2019b],[Palossi et al., 2019a] pioneered fully autonomous navigation for nano-sized drones.

In PULP-DroNet, DroNet [Loquercio et al., 2018], which was previously only deployed on standard-size, commercial drones, was ported to run on the PULP-Shield. The authors present a methodology for porting convolutional neural networks to ultra low-power platforms, while maintaining a reasonable accuracy. They also provide the design of the PULP-Shield, the code

running on the target hardware along with the datasets and trained network, which can be found in this repository.

## 1.5 Our Contribution

While the work on user proximity done on the Parrot Bebop 2 quadrotor was inspiring, the large size of the drone limited the human-robot interaction scenarios. PULP-DroNet targeted nano-sized quadrotors, but it was more about avoiding people than interacting with them. Combining both works was very appealing - we could have a tiny, non-intimidating quadrotor autonomously and closely interacting with users.

The main focus was to deliver a demo, mimicking the behavior describe in [Mantegazza et al., 2019], but on an autonomous nano-drone rather than a standard-sized, remotely-controlled quadcopter. Trying to merge these two concepts, human-machine interactions and ultra low-powered autonomous platforms, proved to be challenging.

We initially tried to use previously collected data recorded by [Mantegazza et al., 2019]. It was recorded using a high-fidelity, full HD camera. The models trained on that data did not generalize well to images captured by the low-fidelity, miniature camera mounted on the nano-drone. We explored various image processing techniques to augment the data, trying to transform the original images to look like images captured by the target camera. This step improve the accuracy of the model slightly, but it was not yet satisfying. In order to perform well on the low quality images we collected an ad-hoc dataset, using the target camera. This led us to build, with drills, 3D printers and code, our data acquisition infrastructure. We also faced the difficulty of collecting data during a pandemic. The people in the office were scarce to none, which required us to bring extra sets of clothes from home, experiment with hair styles, hats, bandanas and face masks. It also developed our skills of simultaneously operating the capture equipment and acting as the subject. This newly acquired data was augmented with the image processing techniques we implemented earlier, and models trained on this data generalized very well to unseen images from the low-fidelity camera.

The main component of our hardware is an academic chip. It was exciting to be among the first to work on it, and this work included tackling the challenges of a project that is not fully mature. The required hardware was not always accessible to us, and what seemed like a disadvantage, proved to work in our favor. By verifying our work on emulators, we were able to identify unexpected, and sometimes unwanted, behaviors on the actual hardware.

Running CNNs on resource-constrained hardware involves a lot of difficult tasks: usage of low-level interfaces, utilization of computational resources by parallelization, neural network minimization and memory management acrobatics. To handle these tasks, we rely on an existing software tools. These tools were the product of an ongoing research and active development. A significant part of our effort was devoted to extending and improving the existing software by communicating our needs to the authors of these tools, and assisting them with debugging. This collaboration was mutually beneficial as we produced a successful demo and the software tools evolved into a more mature utility.

In general, the work revolves around multiple hardware devices and even a larger range of software libraries. Time and effort was dedicated to learning how to work with them, integrating them and writing common interfaces when possible. The ability to switch between different variants of our target platform without having to change the code extensively was given significant importance.

While we set our eyes on implementing a fully functional demo, we also took a side quest to analyze factors that can impact the quality of deployment in terms of accuracy, performance and energy efficiency. So more than describing the minimal pipeline required for deployment, we also provide insights into how to tailor the process to meet more specific goals. We explored different types of data augmentations that can increase the robustness of models. We discuss considerations such as input size and resizing strategies and their influence on accuracy and performance. We also motivate our choice of neural network architecture, and examine the energy-related consequences of memory-heavy models. That details of this exploration should provide the readers some intuition about where they should focus their efforts, depending on their goals.

## 1.6 Thesis Outline

This document consists of 5 chapters, which are briefly described here:

- **System Description** Chapter 2 - This chapter describes the hardware used for data collection and deployment, and all additional software tools that were used in the process.
- **Solution Design and Implementation** Chapter 3 - A detailed explanation of the various stages of the porting workflow, from the initial model selection to deployment on the target platform.
- **Evaluation** Chapter 4 - We test our solution both online and offline, qualitatively and quantitatively. Analyzing the results, we verify that our objectives were achieved.
- **Exploration** Chapter 5 - In this chapter, we slightly stray from the main quest of porting user tracking capabilities to the target platform, and examine a series of factors that play part in the trade-off between accuracy, performance and energy efficiency.
- **Conclusion and Future Work** Chapter 6 - Here we discuss how to further improve and expand our work.

# Chapter 2

# System Description

We build upon previous work done in the fields of robotics, human-machine interaction, deep learning and embedded platforms. We look to enable artificial intelligence on edge devices, specifically nano-drones, for the purpose of exploring close-range interaction with users. Specifically, we are interested in porting the tracking capabilities of ProximityNet from [Mantegazza et al., 2019] to an ultra-low power platforms. As we relied heavily on hardware and software that was developed previous to this work, we take the time to introduce it in the chapter. We first present the hardware components that form our deployment target: the nano-drone, the low-fidelity camera and the system on chip. These three ingredients are available in two variations - the research-oriented PULP-Shield and its commercialized version, AI Deck. Second, we discuss the hardware and software setup that was used for collecting data, and was essential for the quantitative evaluation of our work. Lastly, we introduce the software tools we utilize during the training, testing and deployment stages.

## 2.1 Hardware

The deployment hardware consists of 3 components - a nano-sized quadrotor, low-power greyscale camera and an ultra-low-power processor. Two different products were tested, a commercial and a non-commercial variations of the same system.

### 2.1.1 Crazyflie 2.0

The Crazyflie 2.0 [bitcraze] is a nano-sized quadrotor, which is open-source and open-hardware. It weighs only 27g, with maximum takeoff weight of 42g. The Crazyflie 2.0 (Fig. 2.1) is 9cm in diameter, making it a perfect candidate to operate in small, closed environment, and populated areas. With fully-charged battery it can fly up to 7 minutes. The Crazyflie 2.0 equipped with a STM32F405 micro-controller, and an additional secondary micro-controller for radio and power management. Its inertial measurement unit includes 3-axis gyro, 3-axis accelerometer, 3-axis magnetometer and a high precision pressure sensor. The Crazyflie 2.0 supports expansion boards, and offers several expansion interfaces such as SPI, UART, I2C and GPIO. The Crazyflie is a popular choice because it is a well-maintained commercial product, which is also open source and benefits from the contribution of the developers community. In addition

Figure 2.1. Crazyflie 2.0 by Bitcraze

to a radio based API for commands, the Crazyflie can be programmed on-board due to it open source firmware.

### 2.1.2 Himax Camera

The HM01B0 [Himax] is an ultra low power CMOS image sensor, and is a attractive option for edge devices due to its energy-efficiency. The HM01B0 contains 320 x 320 pixel resolution and supports a 320 x 240 window mode which can be readout at a maximum frame rate of 60FPS, and a 2×2 monochrome binning mode with a maximum frame rate of 120FPS. The video data is transferred over a configurable 1 bit, 4bit or 8bit interface with support for frame and line synchronization. The sensor integrates black level calibration circuit, automatic exposure and gain control loop, self-oscillator and motion detection circuit with interrupt output to reduce host computation and commands to the sensor to optimize the system power consumption. The Himax (Fig. 2.2) only consumes up to 4mW at a resolution of 320x240 pixels, which places it as the lowest power consuming camera in the industry.

### 2.1.3 GAP8v2

The PULP (Parallel Ultra-Low-Power) platform [ETH/UNIBO] is a multi-core platform developed in ETH and University of Bologna (UNIBO). It was designed to enable computations on edge devices, while remaining energy-efficient.

The GAP8v2 is a RISC-V based PULP parallel architecture that is tailored for DSP and ML applications, commercialized by GreenWaves Technologies. Unlike other commercial microcontrollers, the GAP can provide the computational resources needed to support a complex task such as autonomous flight, which requires workloads in the order of 100 million – 10 billion operations per second [Loquercio et al., 2018]. This platform is at the heart of our work, executing the neural network inference, and due to its importance is reviewed here in detail.

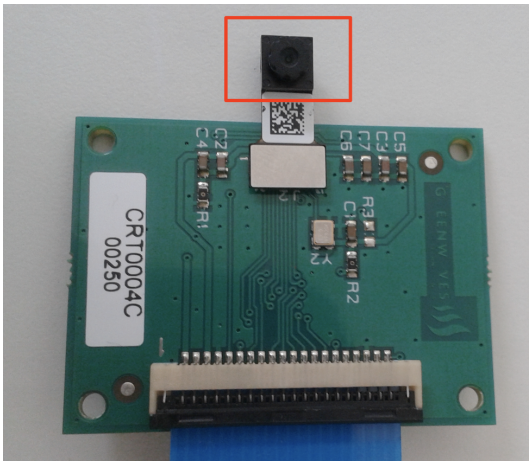


Figure 2.2. Himax ULP camera

This multi-core architecture (Fig. 2.3) is divided to the fabric controller and the cluster, which consists of 8 cores. The fabric controller core is used for control, communications and security functions. The fabric controller acts as master for the cluster, dispatching tasks for it to compute. The cluster is off by default and must be switched on and mounted by the fabric controller in order to be used. Within the cluster, there is one core who serves as an inner master, distributing the work between the 8 cores. The cluster architecture is optimized for the execution of vectorized and parallelized algorithms combined with a specialized Convolutional Neutral Network accelerator (HWCE). All 8+1 cores are high performing, with extending RISC-V ISA. The fabric controller can operate at (150 MHz @ 1.0V; 250MHz @ 1.2V) and the cluster cores at (87 MHz @ 1.0V; 170MHz @ 1.2V). GAP8v2 does not have a floating point unit, which proves a challenge as neural networks use real-valued numbers. To solve this issue, it is possible to implement floating point operations purely in software, but it would be extremely costly in terms of energy and performance. The other solution is using quantized neural networks, which is the method chosen for our work.

### ISA Extensions

The fabric controller and cluster supports the following instructions:

- RV32I Base Integer Instruction Set (full)
- RV32C Standard Extension for Compressed Instructions (full)
- RV32M Standard Extension for Integer Multiplication and Division - Multiplication only.

To optimize further the targeted algorithms, the ISA is extended for:

- Misaligned memory access
- Pointer post/pre-modified memory accesses
- Zero overhead hardware loops
- Bit manipulations - single cycle insert/extract

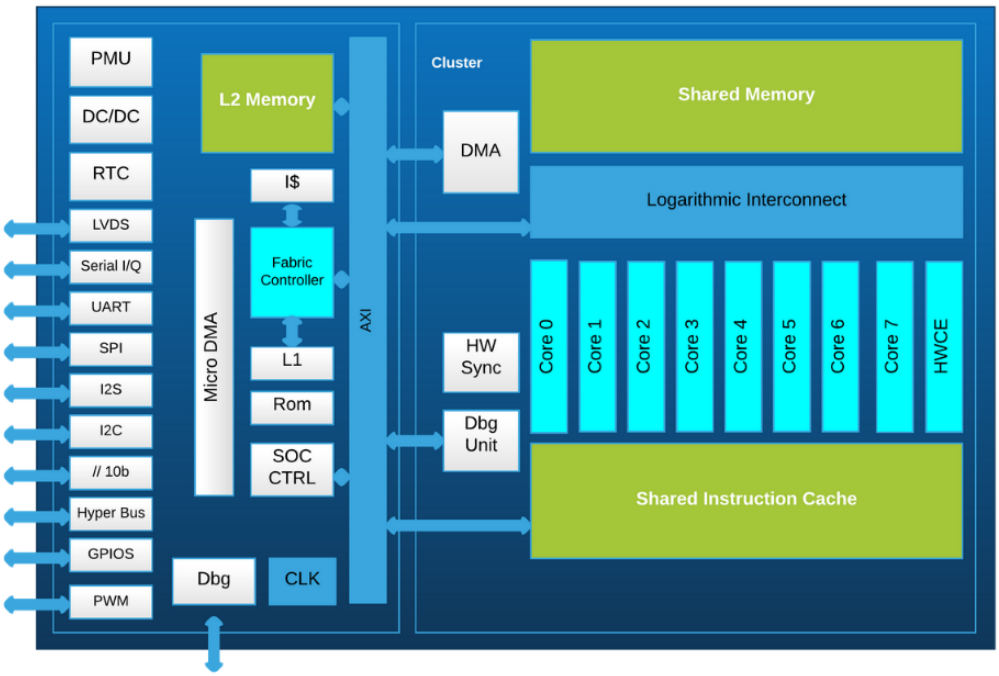


Figure 2.3. GAP8

- Instructions mixing control flow with computation (min, max, etc)
- Vector operations
- Fixed point operations
- MAC instructions

Many of these extensions are tailored for DSP algorithms. Operating on matrices (e.g. images) is a fundamental part, and the vector instructions are set to accelerate these computations. Loops are also a very common element in processing images and signals. Every iteration of the loop we need to advance the pointers, which translates into add instruction, but extensions like auto-increment load/store eliminates the need for this extra instructions. Another important aspect of loop handling is checking each iteration if we comply with the loop condition. This results in branching which is expensive, and the shorter the loop is, the more expensive it is, because we do very little work before we branch. Using hardware loop, we encode the number of iterations we want to make, and avoid the branching. In the Gap8 we have 2 hardware loops. In case we have 3 nested loops in the code, the outermost one would be done with branching and the inner 2, with hardware loops.

As the GAP8 does not contain a floating point unit, fixed-point representation is used for real numbers. Fixed-point arithmetic requires ISA support for this data type, as fixed-point operations can produce results that have more digits than the operands. MAC and dot product operations are also extremely common in scientific and engineering applications, and the extended ISA provides enhanced performance for this purpose.

### Memory Scheme

The memory hierarchy consists of:

- L2 memory of size 512KB
- L1 with 64KB shared by all the cores in cluster
- L1 memory with 8 KB owned by fabric controller

In order to keep energy consumption low, and to avoid maintaining cache coherency, there is no data cache. There is a instruction cache for the fabric controller and a shared i-cache for the cluster cores. The shared L1 is banked and connected to the cores using a logarithmic interconnect, that allows a core to access its bank in a single cycle in 98% of the cases. The L2 memory is shared all processing units and the DMAs.

L2 is divided into private banks, and interleaved banks. In the private banks, data is stored continuously, row-by-row. In the interleaved banks, data is stored bank-by-bank (in the depth axis), so the first row is written to the beginning of bank 0, but the second row is written to the beginning of bank 1, and the third to beginning of bank 2 and so on. Since we do DSP, we regularly access vector data. By using interleaved banks, we allow different cores to access the same data with minimal stall, and minimize the probability of bank conflict. This is roughly emulating multi-port memory cards, where there is no stall at all.

Offloading memory transactions to the DMAs allows the fabric controller to perform other computations:

- $\mu$-DMA - This subsystem handles access to peripherals. Since the chip is meant to support several sensors, it might need to provide a bandwidth of up to 1Gbit/s. There is no MCU in the market that can handle such bus traffic. The $\mu$-DMA is an autonomous core that speaks the language of the various peripherals,and handles data transfer from/to L2. The $\mu$-DMA can be used to access external storage (L3) over the HyperBus, like Flash or RAM. Once the transfer is done the fabric controller is receiving an interrupt from the $\mu$-DMA. There is hardware support for double buffering which allows continuous data transfers. 8, 16 or 32-bit transfer width is supported, and up to 128kB can be transferred in a single transaction. Can queue up to 2 transfer requests.
- Cluster DMA - Multi-channel 1D/2D DMA unit which is used to transfer data between L2 and shared L1 of the cluster cores. Can queue up to 16 transfer requests.

### Peripherals

The Gap8 supports a variety of peripheral interfaces. We mention below the ones that were used in our work:

1. Inter-Integrated Circuit ($I^2C$) - A synchronous serial communication bus widely used for low-speed peripherals. It was used to configure the camera registers
2. Camera Parallel Interface (CPI) - 8-bit wide interface used to receive frames from the camera.
3. HyperBus - 8-bit wide, high speed memory bus that connects to HyperRAM/HyperFlash devices. This was used for memory transfers between L2 and L3.

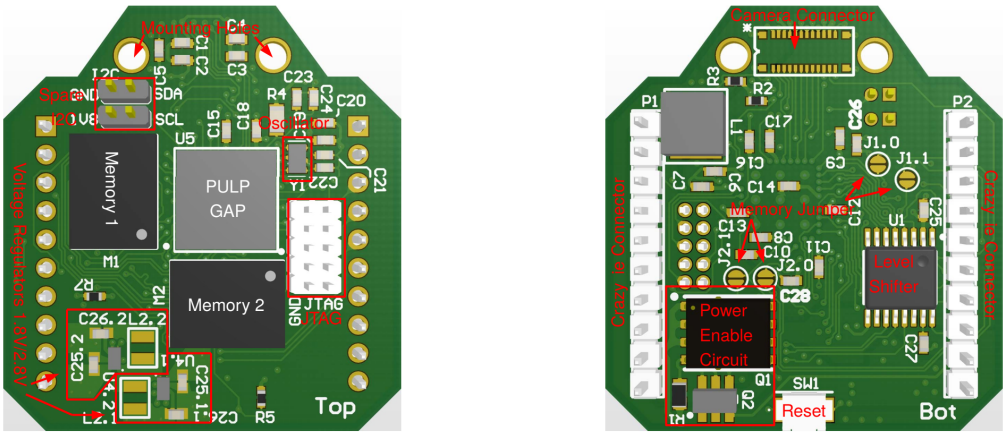


Figure 2.4. PULP-Shield. Left: Top view. Right: Bottom view

4. Serial Peripheral Interface (SPI) - A synchronous serial communication interface . This interface used used to communicate with the firmware from the GAP8.

5. Universal Asynchronous Receiver/Transmitter (UART) - A hardware device used for asynchronous serial communication. This interface used to communicate with the firmware from the GAP8.

6. General-purpose input/output (GPIO) - A digital signal pin whose behavior can be controlled by the user at run-time. These pins were used to light up LEDs on the nano-drone to indicate the status.

7. Dbg - Interface provided by the manufacturer for debugging the application on-board.

### 2.1.4 PULP-Shield

The PULP-Shield [Mueller, 2018] is a compact printed circuit board, which can be plugged to the target nano-drone, the Crazyflie 2.0. The printed circuit board (Fig. 2.4) includes 2 SPI interfaces, one directly connected to the Himax ULP camera, and a second connecting the accelerator to the existing micro-controller. The processor in the heart of the printed circuit board is GAP8v2. The two memory slots are mounted with a 8MB RAM and 16MB Flash memory. The board includes mounting holes for a camera holder, that hosts the Himax camera. The PULP-Shield's for factor is 30x28mm, and it weighs only 5g, putting it well below the Crazyflie's payload limit of 15g. The micro-controller of the Crazyflie is tasked with the real-time kinematics estimation and the flight controller, while GAP8v2 SoC serves as the visual navigation engine. Extending the Crazyflie with the PULP-Shield makes it possible execute ML algorithm on-board. The interaction between the components in the system, during inference, are as follows:

1. The host wakes the GAP8v2 accelerator with an init interrupt.

2. Binary is loaded from the external Flash memory to the L2 of the GAP8v2, via the HyperBus, and executed.

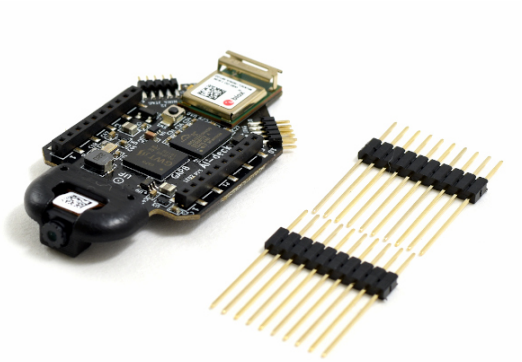

Figure 2.5. AI Deck by GreenWaves Technologies

3. The Himax camera is configured using the I2C interface.
4. Frames grabbed from the camera are transferred to L2 by the $\mu$-DMA.
5. Additional data required for inference, like the neural network's parameters, can be loaded from the DRAM/Flash memory via the HyperBus.
6. Parallel execution begins on the cluster
7. The results are communicated to the Crazyflie micro-controller via SPI.

### 2.1.5 AI Deck

The AI Deck [Technologie] is a commercialized version of the PULP-Shield, with additional WiFi connectivity functionality that enables streaming of images to a personal computer. The communication between the host and the accelerator is done via UART instead of SPI. It is presented in Figure 2.5

### 2.1.6 Gapuino

This is an Arduino Uno form factor board which includes the GAP8 processor. It is a commercial product manufactured by GreenWaves Technologies. It comes with a Himax camera and a larger external memory than the PULP-Shield and AI Deck. Unlike the PULP-Shield and AI Deck it can be powered straight from the laptop using a USB-B connection, so it does not require an additional power source. All communication is done through the USB-B cable with no need for the JTAG. This product is sturdier than PULP-Shield and AI Deck and is more suitable for non-flight data collection.

### 2.1.7 Parrot Bebop 2

The Parrot Bebop 2 [Parrot] is a light-weight drone, designing primarily to capture full HD video. With weight of 500g, it is considered a standard-sized drone, a few orders of magnitude larger than the Crazyflie. Flight time is up to 25 minutes on a fully-charged, undamaged battery, which is very convenient for data collection. The Bebop (Fig. 2.6) is also quite powerful, it can

Figure 2.6. The Parrot Bebop 2

speed up to 37.28 mph horizontally and 13.05 mph vertically, while maintaining the image quality. The Parrot Bebop 2 was used in the work of [Mantegazza et al., 2019], and also in the data collection stage of our work. Unlike the Crazyflie, the Bebop is not programmable, but it can be controlled remotely using WiFi. The camera is equipped with a 14 megapixel, wide-angle lens, which offers a field of view of $180^o$ horizontally and vertically. The Bebop includes a video stabilizer, which crops a virtual window of 1920x1080 pixels such that the roll and pitch of the drone are compensated. There is a stark difference between this high quality camera and the Himax. It benefits from higher resolution, better auto-exposure algorithm and the implemented video stabilization. Despite these advantages, the cropped field of view is $77° \times 49°$, smaller than the Himax's $82° \times 65°$ field of view. As previous work has shown, the task of following a user can be accomplished with the Bebop's field of view, suggesting it captures enough information about the scene. This indicates that the Himax, enjoying a larger field of view, should also be able to handle the task.

### 2.1.8 OptiTrack

The data collection took place in the Drone Arena, a 10x10 $m^2$ room, equipped with a motion capture system by OptiTrack [OptiTrack]. The system is mounted on a rig attached to the ceiling, and includes 12 cameras, with RGB and IR sensors (Fig. 2.7). The tracked area is 6x6 $m^2$, but for safety reasons the drones are limited by a virtual fence to a smaller area of 4.8x4.8 $m^2$. The IR sensor are used primarily to track the markers (put an image). Markers are placed on the drone and on the user during data collection and quantitative evaluation. The configuration of IR-reflective balls is different for the drone and the user, allowing us to distinguish between the tracked poses. The system is accompanied by a software tool called Motive, where the trackers are defined. Motive also handles the calibrations process, and provides an interface to control system parameters, like tracking rates.

## 2.2 Software

### 2.2.1 ROS

The Robot Operating System (ROS) is a set of software libraries and tools that helps in creating a robotic application. It is a industry and research standard for framework for robotics. Although

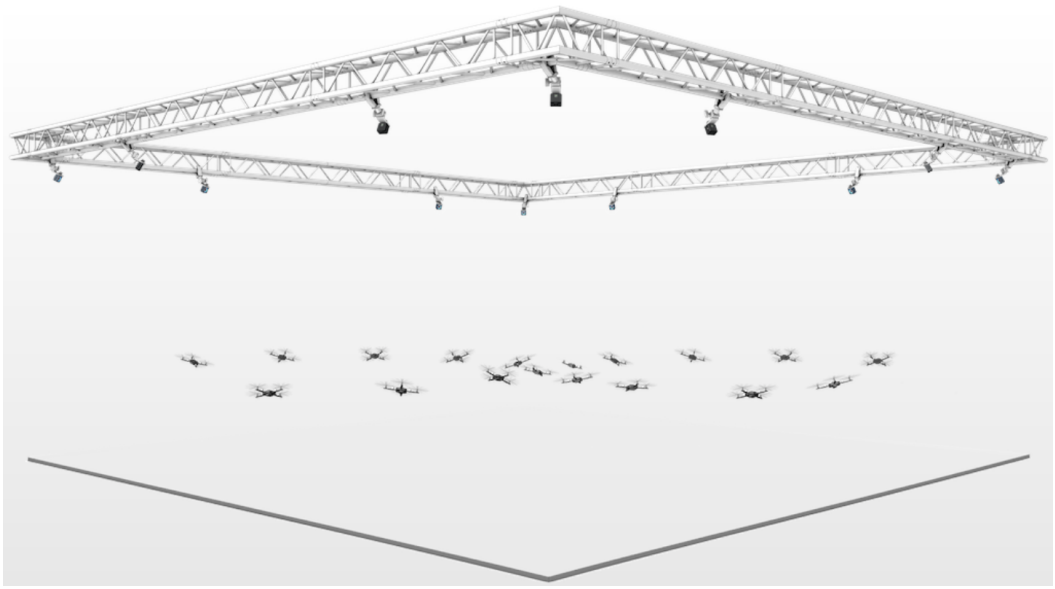

Figure 2.7. An example of OptiTrack configuration with 12 cameras

ROS is not an operating system, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

ROS was employed to aid in the data collection phase. Using the framework, video from several sources and pose information from the motion capture system were time-stamped, collected and stored in a format that is easy to handle. This is done using ROS bags, a logging format for storing ROS messages. Files recorded using this format can be played and edited using command line tools and by writing ROS specific code in C++ or Python. Topics of interest, like video frames or pose can be published using ROS nodes, and recorded with the ROS bag package.

Two ROS-based repositories were used during this work. crazyflie ros is a ROS stack for Crazyflie which allows, among other services, message publishing and configuration of the drone's parameters. It is described extensively here [Hönig and Ayanian, 2017]. drone arena is a package that provides common API to all drones used in the Drone Arena. It allows the system operator to communicate a set of known commands to the drone, such as take off, velocity control and flying to a selected coordinates. It also implement safety protocols like the virtual fence, landing when the battery is low and switching off the motors after a crash.

### 2.2.2 PyTorch

PyTorch is an open source machine learning library based on the Torch library. PyTorch was used in the training, quantization and off-board testing phases.

### 2.2.3 Numpy

NumPy is the fundamental package for scientific computing with Python. The library provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. Numpy was used extensively in the project, in all off-board applications, from data collection to evaluation and visualization.

### 2.2.4 OpenCV

OpenCV is a library of programming functions mainly aimed at real-time computer vision. OpenCV core is open-source. OpenCV is used extensively in industry and research. The library was used in all off-board stages for image manipulating, including:

- Resizing
- Cropping
- Converting between color spaces
- Data augmentation
- Visualization
- Camera calibration

### 2.2.5 Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. The library was used for validating data collection correctness, and analyzing the results of both quantitative and qualitative evaluations.

### 2.2.6 Pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. Pandas were used to package and read the datasets used for training, validation and testing.

### 2.2.7 NEMO - NEural Minimizer for pytOrch

NEMO [Conti, 2020] is a small python library that minimizes PyTorch deep neural networks, to enable their deployment on memory constrained low-power platforms. As the GAPv2 has no support for floating point operations, the trained model had to be quantized in order to be deployed, and NEMO was used to convert full precision models to 16-bit or 8-bit fixed point. The first generation of NEMO produced the quantized weights as text and .hex files, with 16-bit elements. The second generation outputs an .onnx file, which describes the model architecture, additionally to the weights (Fig. 2.8).

A DNN layer is treated as composition of operators from 3 different classes: linear (convolutions, fully connected), batch normalization and non-linear activations (ReLU). NEMO supports 4 DNN representations, which are discussed below.

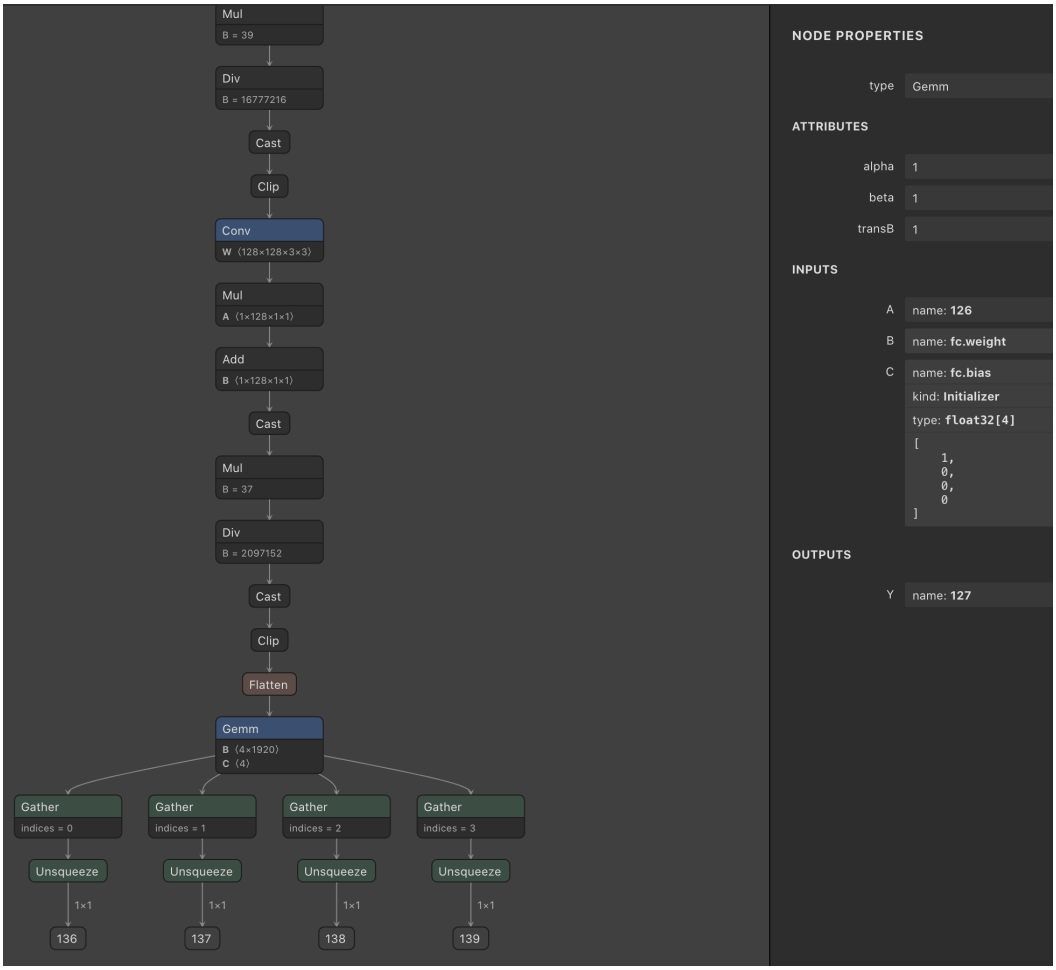


Figure 2.8. A visualization of the .onnx model using Netron

Full Precision

This representation is the standard, real-valued neural network that is equivalent to the a PyTorch model under some restrictions - in NEMO branches starting from a layer that is not an activation layer are forbidden.

FakeQuantized

A tensor $t$ is called quantized if for all elements $t_i \in t$

$$t_i = \alpha_t + \epsilon_t \cdot q_i, q_i \in \mathbb{Z}_t \tag{2.1}$$

Where $\epsilon_t \in \mathbb{R}$ is called quantum, $\alpha_t \in \mathbb{R}$ is called offset, and $\mathbb{Z}_t$ is a subset of $\mathbb{Z}$ which is referred to as the quantized space. To compute the quantized version of $t$,we need to find a mapping $Q_t : \mathbb{R}^D \rightarrow \mathbb{Z}_t^D$ such that

$$\hat{t} = \alpha_t + \epsilon_t \cdot Q_t(t) \tag{2.2}$$

The mapping $Q_t(t)$ is called the integer image of $t$, and the final goal of the quantization process is to use $Q_t(t)$ instead of t, without compromising the accuracy. The cardinality of the set $\mathbb{Z}_t$ will determine how many bits are required to represent it. In the FakeQuantized representation weights of linear operator and outputs of activations are real-valued but are limited to a set of restricted quantized values during inference. By using PACT on ReLU activations, we clip the output value to be in range between 0 and a maximum value of $\beta$, and the activation uses the quantum $\epsilon_t$. To represent a tensor $y$ with Q bits and range of $[0, \beta_y]$, we need $\epsilon_y = \beta_y/(2^Q-1)$. Linear weights are stored in full-precision but are similarly clipped using PACT-like quantization [Choi et al., 2018] during a forward pass.

$$\hat{w} = \left\lfloor \frac{1}{\epsilon_w} \text{clip}_{[\alpha_w, \beta_w)}(w) \right\rfloor \cdot \epsilon_w \tag{2.3}$$

In this representation, real tensors are considered and updated using the back-propagation step, so training is similar to that of full precision models, as gradients are computed on full-precision tensors ignoring the quantization functions. This version of the network is obtained by executing:

```
net = nemo.transform.quantize_pact(net, dummy_input=dummy_input)
```

The batch normalization layers are used during the training stage to normalize the inputs to a layer for each mini-batch. It is known to speed up the training process and improve the performance of the model [Ioffe and Szegedy, 2015]. The batch normalization layers ensure that the dynamic range of the activations is bounded, which aids with the quantization process. Batch normalization layers by quantization-aware substitutes using a fixed point format defined by the required precision and range. During inference, the batch normalization layers can be merged with the previous convolutional layer in a process called folding, which can be performed during the FakeQuantized step

```
net.fold_bn()
net.reset_alpha_weights()
```

QuantizedDeployable

The FakeQuantized representation is suitable for training and fine-tuning, but it is not a an integer-only quantized neural network. The quantization was defined for weights and activations, but not for all intermediate representations. The QuantizedDeployable representation completes the final step in quantizing the neural network, by making sure all the computations are performed on quantized inputs, and produce quantized outputs. Transforming a model into a QuantizedDeployable representation in NEMO requires 3 steps:

1. Quantizing the batch-normalization layers

   ```
   net = nemo.transform.bn_quantizer(net)
   ```

2. Freezing the linear weights in their quantized form

   ```
   net.harden_weights()
   ```

3. Propagating the quantum $\epsilon$ along the network

   ```
   net.set_deployment(eps_in=1./255)
   ```

IntegerDeployable

As all quantized tensors have an integer image $Q_t(t)$, their real-valued nature can be discarded by only considering their integer images. This step results in the IntegerDeployable representation. However, NEMO still stores data as floats in all representations.

To convert a model to IntegerDeployable representation in NEMO, all parameters are replaced by integer equivalents and several operators are changed.

```
net = nemo.transform.integerize_pact(net, eps_in=1.0/255)
```

There are two quantized spaces, $\mathbb{Z}_t$ for the weights and $\mathbb{Z}_y$ for the activations output. The Quantization/Activation operator is responsible for both providing the non-linearity functionality vital for deep neural networks, and squashing the input tensor $t$ into the quantized space $\mathbb{Z}_y$. Requantization is used when transforming a tensor from one quantized space to another. This procedure is not trivial, and solved as an approximation using information computed in the IntegerDeployable step.

### 2.2.8 AutoTiler

The AutoTiler is a tool that allows the developer to focus on writing basic kernels for tasks that will be executed on the cluster, without worrying about complying with the L1 memory constraints. To use the Autotiler, the layers must be codded manually in c and then compiled with the Autotiler. The code for these manually written kernels describes the functionality of each layer. However, it does not directly handle the complex memory management involved in an extremely memory-constrained system such as the GAP8. Our computations are done on elements in the memory closest to the processor, L1, but since it is a small memory, not all data can fit there. This means we need to break down our input into smaller pieces, and orchestrated

the transfers between the bigger memories and L1 in the most efficient way. The AutoTiler is a tool that abstracts these memory constraints from the programmer, generating a tiled structure that complies with the memory budget. It is run on the user's computer before the code is compiled, and generates GAP8-specific code. The tiled structure is used to transfer data across the memory hierarchy in a pipelined way that hides memory latency and ensured the cores are not idle. As we are dealing with fixed point operations, a scheme must be implemented to handle the possible information loss involved in these calculations. One important aspect of handling it is using accumulators - variables that store intermediate results using more bits than the operands. However, the fixed point accumulator implemented in the AutoTiler has certain limitations. We lose accuracy after each operation, and as we go deeper into the neural networks, we diverge from the non-quantized neural networks. To avoid this divergence, we can configure NEMO to shrink the dynamic range by discarding outliers. Large-values activations with low probability are ignored, and in turn we can spare more bits for the fraction part, decreasing the numerical instability. Data is handled as one dimensional array, and the layout used is CHW (channel, height, width). If our tensor is an RGB image, it means the elements are stored as RGB triplets for every pixel. For a HWC layout, the memory would look like a sequence of three images, each image corresponding to one color channel. As we switched to DroNet architecture at that point, it was possible to reuse the code written for the PULP-DroNet, with small adjustments, and it was not required that we write kernels from scratch.

### 2.2.9 DORY - Deployment ORiented to memorY

DORY [Burrello et al., 2020] is a python library for deploying deep neural networks on low-cost micro-controllers, that generates ANSI-C code. It solves a constraint programming problem, by maximizing the L1 memory usage subject to the topological constraints introduced by the neural network's architecture. Dory, much like the AutoTiler, abstracts the tiling from the programmer. However, unlike the AutoTiler which uses CHW data layout, DORY uses PULP-NN as the back-end, which is based on a HWC layout. Tensors are tiled to comply with the memory constraints, and transfers between memory levels are managed by the library. To maximize performance double buffering is used, and while the current tile is being computed, the DMA is used to fetch the next tile. DORY targets devices with 3 levels of memory hierarchy (L1, L2, L3). L1 is a small and fast memory that is closest to the processor. L2 is slightly larger, but slower, and when an data element cannot be found in L1, it is retrieved from L2. L3 is an external memory, such as Flash or RAM, which is slower and significantly bigger than L2. In DORY, there is no theoretical size limit on the L3 storage size, so it can store the weights of very large deep neural networks. L2 is expected to be of more modest size (up to a few MB) and have reasonable bandwidth. L1 is assumed to have unlimited bandwidth to the cores, with size smaller than 128kB. This design targets specifically the GAP8 architecture. DORY operates in 3 stages which are discussed below.

#### ONNX Decoder

ONNX decoder - DORY received as input a ONNX-formatted graph describing the quantized neural network. This .onnx file is the output of the second generation NEMO. The graph is reorganized as a set of layers. Weights , inputs and outputs are quantized while temporary data is stored as 32-bit integer.

### Layer Analyzer

Layer analyzer - The first stage of optimization, where layers are considered separately from one another. It includes three components:

1. Tiling solver - the solver first checks whether L3-L2 tiling is required by testing if

$$L2_{\text{w,curr}} + L2_{\text{w,next}} + L2_{\text{i,curr}} + L2_{\text{o,curr}} < L2 \tag{2.4}$$

   Where the w subscript signifies this is a memory storing weights, the i subscript refers to the input buffer, and the o subscript to the output buffer. Buffers storing elements needed for the current layer are notated with 'curr', and memory related to the next layer is notated respectively. The weights of the next layer are considered, due to the double buffering mechanism. If the condition is not satisfied, L3 tiling is performed, partitioning the layer to smaller, identical parts. If the resulting tiles can't satisfy equation 2.4, the algorithm reports an error as only L2-L1 tiling is supported for activations. The next step is finding an appropriate tiling for L2-L1, which is treated as a Constraint Programming problem - maximizing the L1 utilization under topological and memory constraints.

2. PULP-NN heuristics - the objective function maximized in the Tiling Solver is augmented with a set of heuristics tailored for a particular back-end, in the case, the PULP-NN. These heuristics involve load-balancing between the 8 cores, buffer reuse and performing a full computation in the channel dimension. Despite reducing thee solution space considerably, these heuristics boost performance.

3. SW-cache generator - Given a tiling solution, this component generates a c code that coordinates all operations required for later execution, such as data transfers and computations. By using asynchronous, pipelined data transfers, the memory latency is mostly hidden.

### Network Parser

After the Layer Analyzer completed the layer-wise tiling, DORY exploits information obtained from all layers to compose a network graph, where every layer is a callable function. In this stage, DORY automatically generates network-wise code, that defines the inference process by combining the individual per-layer code produced by the SW-cache generator. For each layer, the weights are fetch from L3, the layer is executed and the input and output buffers are manages. DORY allocates two memory buffers, for L2 and L1. The overall required sizes for these memories are computed offline, taking into consideration residual connections. The execution pipeline as orchestrated by DORY can be viewed in Figure 2.9.

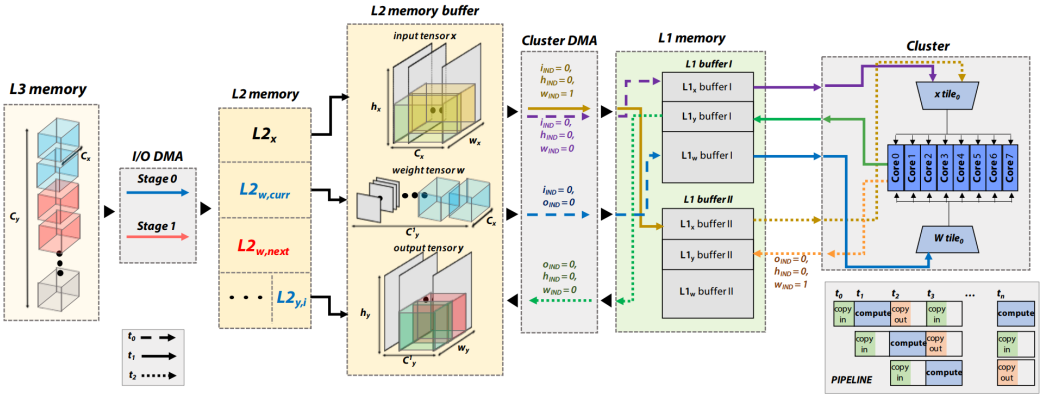


Figure 2.9. L3-L2-L1 layout, from the paper [Loquercio et al., 2018]

# Chapter 3

# Solution Design and Implementation

It is our goal to propose a pipeline that supports the porting of convolutional neural networks to edge devices. Specifically, we want to perform the user tracking task of [Mantegazza et al., 2019] on our target platform, a Crazyflie nano-drone extended with a GAP8 chip. We present a process that can be divided into 5 main steps - model selection, data collection, training, testing and deployment. In this chapter we introduce the tools we developed to support this process. The C and Python code for data acquisition, training, testing and quantization can be found in this GitHub repository. These tools are used for evaluation of a deployed model in Chapter 4 and a more exhaustive exploration in Chapter 5.

The development process has been an iterative one, where exploring required repeating most, if not all, the stages of the pipeline. Selecting a new architecture for our convolutional neural network is a rather obvious example of having to repeat all stages. Another example is recognizing weaknesses in the current model during testing and deployment, which led to collection of entirely new datasets, and again compelled us to re-train the model, verify it on the test-sets and repeat the deployment. On occasion, difficulties we encountered on the last stage of deployment forced us to return to square one, and choose a simpler architecture. In order to complete this iterative work, a flexible software infrastructure was developed to support all stages, ensuring delivery of high performing models with low overhead for the developers.

## 3.1 Models

The common denominator for all models we examined was that they are all convolutional neural networks. First, because they are commonly used for image analysis. Convolutional neural networks became a commodity in computer vision after the success of AlexNet [Krizhevsky et al., 2012]. Second, because the deployment tools supports this class of neural networks. The convolutional kernel is very suitable for execution on GAP8, since the computations are highly parallelizable so we can make use of the cluster, and it mainly consists of multiply–accumulate operations which is supported by the ISA.

These reasons make convolutional neural network a favorable choice for computer vision tasks on edge devices.

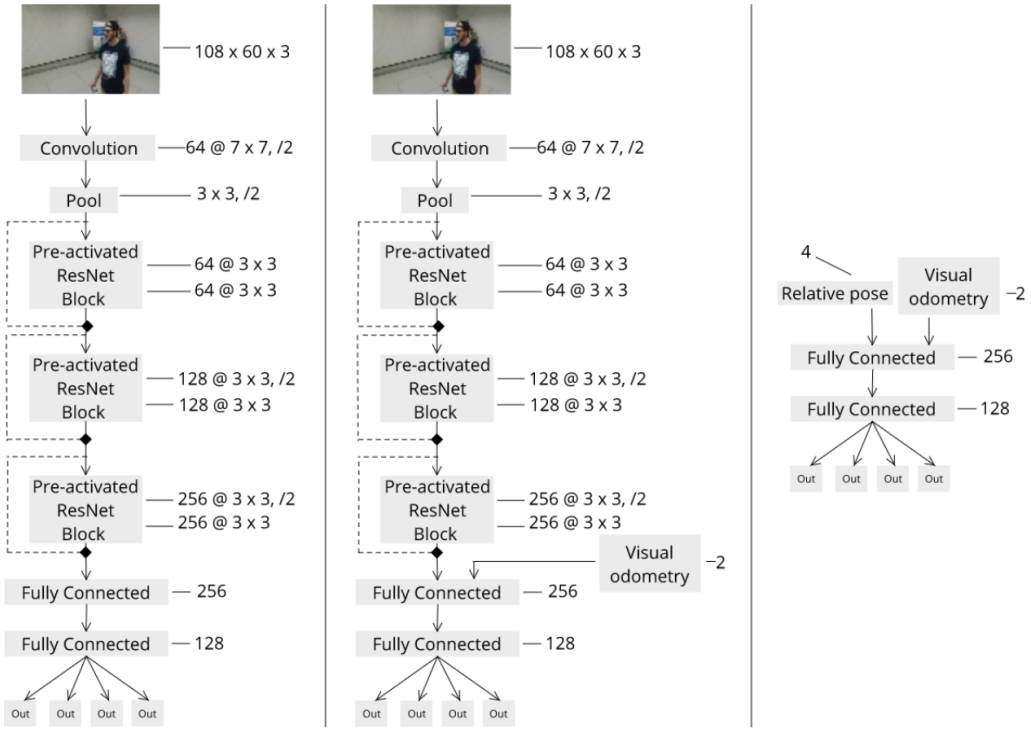


Figure 3.1. Illustration of the 3 models discussed in ProximityNet paper

### 3.1.1 ProximityNet

In the paper of [Mantegazza et al., 2019], the focus on the task of controlling a quadrotor in order to hover in front of a person who is freely moving. The authors proposed two approaches:

1. Mediated - A model maps video frames to states, which are the relative pose of the user with respect to the drone. This is a ResNet-inspired model that receives the image as input, and predicts the relative pose. The prediction is then fed to a controller. The controller can be implemented as a control loop or by training a simple multi-layer perceptron that takes the relative pose and odometry as inputs, and predicts the controls.

2. End-to-End - A similar model to (1) with additional input of odometry, that skips the pose estimation, and directly predicts the control signals. Video frames and odometry are mapped to control output in one step.

In the paper, both approaches yielded similar results. Training of model (2) requires, in addition to video recordings, also the ground truth of the controls. This would require a skilled pilot to fly the drone, or using the output of dedicated controller. To simplify our task, we decided to focus on model (1), which predicts the relative pose.

The code provided by the authors was written in Keras. For our deployment pipeline, we first had to implement it in PyTorch, since it was the framework compatible with the deployment tools. To verify the porting was successful, we first trained and tested a model that receives RGB

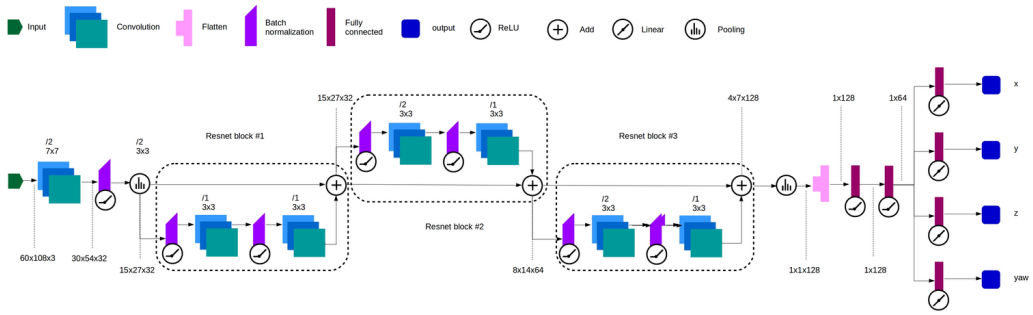


Figure 3.2. Modified ProximityNet architecture

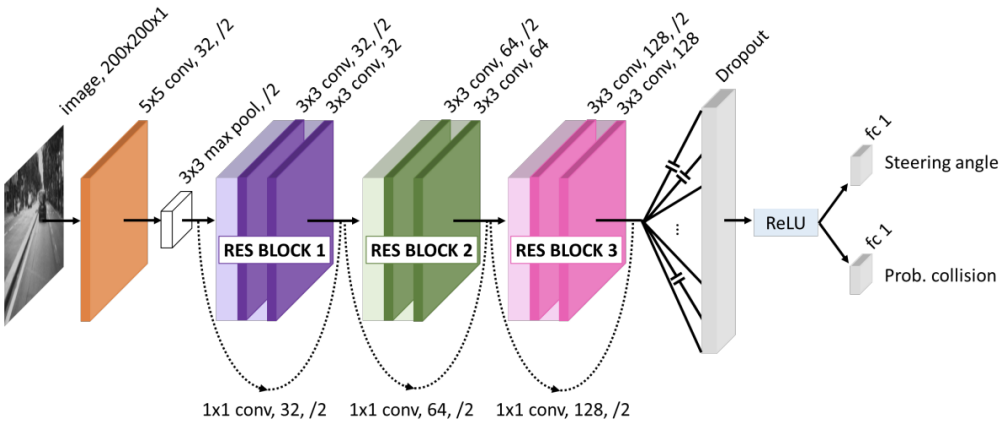


Figure 3.3. Original DroNet architecture [Loquercio et al., 2018]

input, which we can compared to the baseline. Once the results were satisfying, the NN was adjusted to handle gray-scale input, as our target camera, Himax, is grey-scale. In additional, the model's complexity was decreased by using 32 channels in the first convolutional layer, instead of 64. The modified ProximityNet is describe in Figure 3.2. ProximityNet receives a 108x60 image input, and outputs 4 float variables $(x, y, z, \phi)$.

### 3.1.2 DroNet

The DroNet [Loquercio et al., 2018] model is inspired by ResNet [He et al., 2015], which is a trait shared with ProximityNet.

DroNet receives as input unprocessed, grey-scale, 200x200 pixels images, and produces two outputs. The probability of collision with an obstacle, and the desired steering angle, given the visual cues from the camera. Despite having a rather large input size, DroNet has roughly 320,000 parameters, while ProximityNet with an input size which is 6 times smaller, has around 265,000 parameters. This can be attributed to the extra dense layer in ProximityNet, and to the fact that the first convolutional layer has 64 channels, as opposed to 32 in DroNet.

The DroNet model we used is a slight modification of the model (Fig. 3.4) used for the PULP-DroNet paper. Since our objectives are different, instead of two outputs we had four

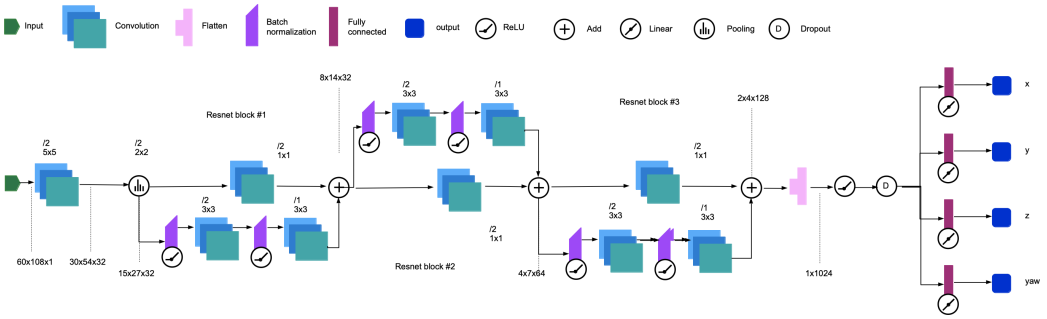


Figure 3.4. Modified DroNet architecture

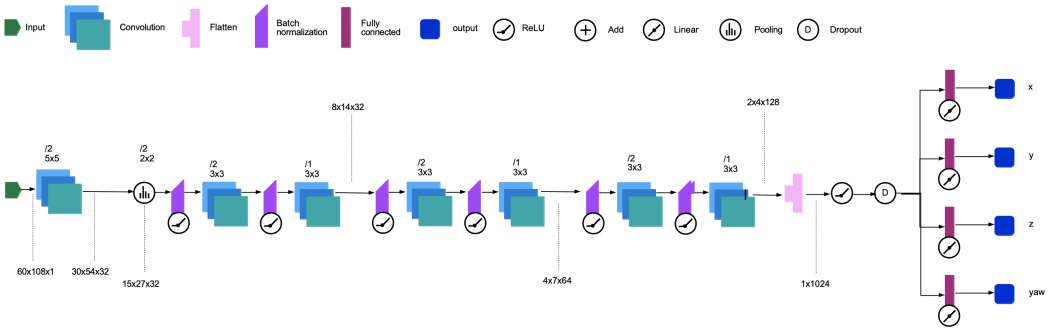


Figure 3.5. PenguiNet architecture

$(x, y, z, \phi)$, all which are regressions, so we had no need for the softmax layer. As the input size also differed, the dimensions of the fully connected layers were affected.

### 3.1.3 PenguinNet

The VGGNet [Simonyan and Zisserman, 2014] offered improvement by using small filters and increasing the number of layers. The PenguinNet (Fig. 3.5) is a VGG-like architecture, with the addition of batch normalization layers, which are known to accelerate the training [Ioffe and Szegedy, 2015]. With PenguiNet we aimed for a simpler model than the two previously mentioned, that will allow us to complete the task with acceptable accuracy. For this reason we removed the residual connections. We decided to keep the batch normalization layers, as they are necessary to bound the dynamic range of the activations, which is beneficial for the quantization process. Also, it is possible to fold them into adjacent convolutional layers. The choice of PenguiNet is discussed in more detail in Chapter 5.

## 3.2 Data Acquisition

### 3.2.1 Camera Calibration

In order to better understand the data we collect, and compare the Bebop and Himax images, we need to know the parameters of our camera. This data was readily available for the Bebop,

| Camera | FoV x | FoV y |
|---|---|---|
| Himax | 82.24 | 65.65 |
| Bebop | 77.85 | 49.4 |

Table 3.1. Field of view for Bebop and Himax cameras

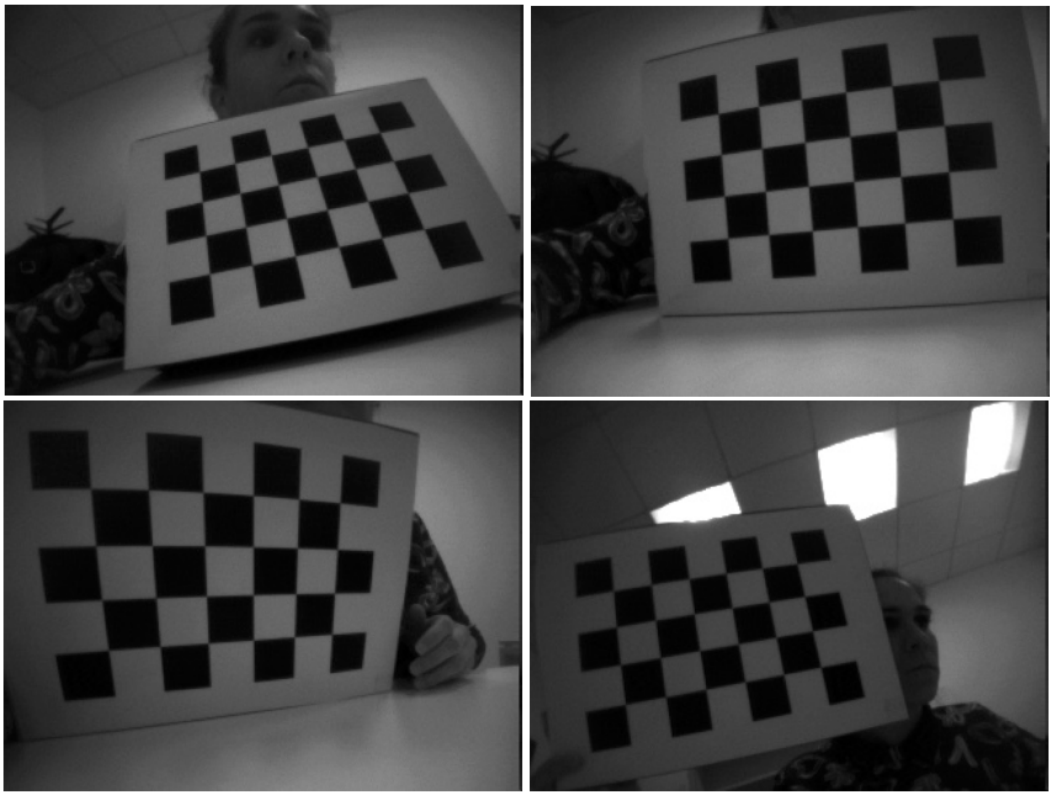

Figure 3.6. Image sequence from the camera calibration process

but not for the Himax. Given that we had more than one Himax camera, and that parameters can vary between cameras, especially on low-end devices, we wrote a calibration utility. The application allows the user to capture chosen images from the Himax, save them to a folder, and run OpenCV calibration, extracting all the relevant information. The calibration code we wrote can be found here, and can be used by running the CameraCalibration.py script. The results of the calibration process are presented in Table 3.1. Sample frames captured during the calibration process can be seen in Figure 3.6.

### 3.2.2 Parrot Bebop 2

Data collection was done exclusively in the Drone Arena, using two cameras, the Himax, which were the discussed above, and the camera mounted on the Parrot Bebop 2. The framework for collecting data from the bebop and the OptiTrack in the Drone Arena was mostly implemented for Dario's thesis. The way in which data is processed, before it is packed in pickle files, was


Figure 3.7. Gapuino platform

modified to account for the target platform.

### 3.2.3 Himax

For Himax data, both PULP-Shield and the Gapuino (Fig. 3.7) platforms were used. Since neither of the platforms has enough memory for data collection, and cannot support ROS bags, the images from the Himax had to be streamed via JTAG to a host computer. First, images must be written to the board's memory, and then transferred through the PULP-bridge to the PC. Since the PULP-bridge is intended mainly for debugging purposes and not for image streaming, the FPS was low. To orchestrate this recording setup, I wrote c code to run on the embedded device, which captures the images from the Himax camera and writes them into a named pipe on the host (using the bridge). On the host, a ROS node reads from that pipe, and publishes it as ROS image messages that can be recorded and viewed. While the images are captured on the device, they are stamped when they reach the host. We account for this delay, in order to sync the video with the OptiTrack pose information. During the acquisition process, we noticed the Himax is very sensitive to the IR light from the OptiTrack system, and it results in very bright spots on the image. For this purpose we purchased IR cut filters, to avoid these artifacts.

### 3.2.4 Acquisition Setup

Initial data collection focused on getting familiar with the Himax, in which a simple setup was used. we wanted to analyze the Himax images vs. the Bebop ones, and it was important that we can capture the same scene from both cameras.

The initial setup was quickly improvised with cardboard, bubble wrap and a hair band (Fig 3.8). However, it was hard to move it around, so a more robust structure was built for mounting both the Gapuino and the Bebop (Fig. 3.9).

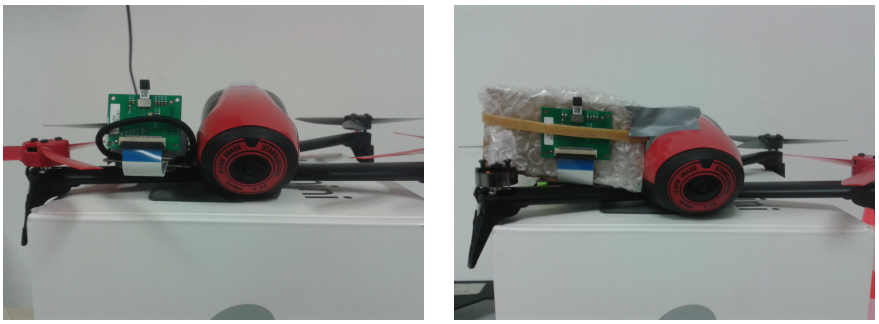

Figure 3.8. Initial acquisition setup

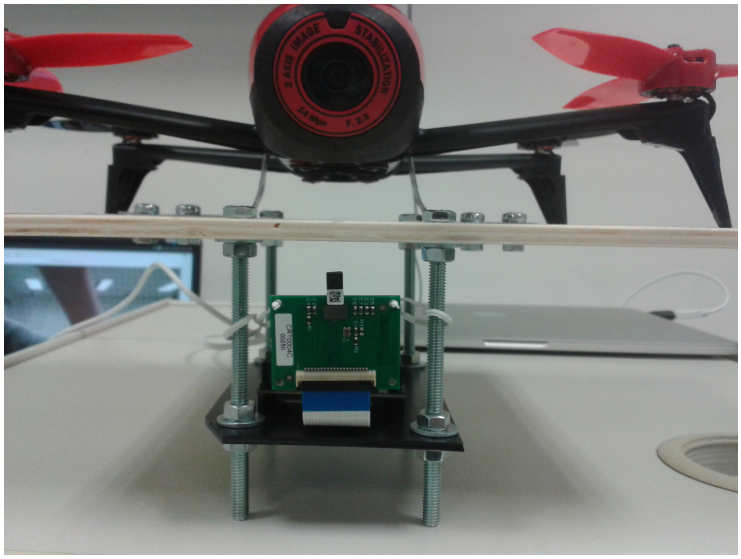


Figure 3.9. Wooden structure for mounting the cameras, and keeping then aligned

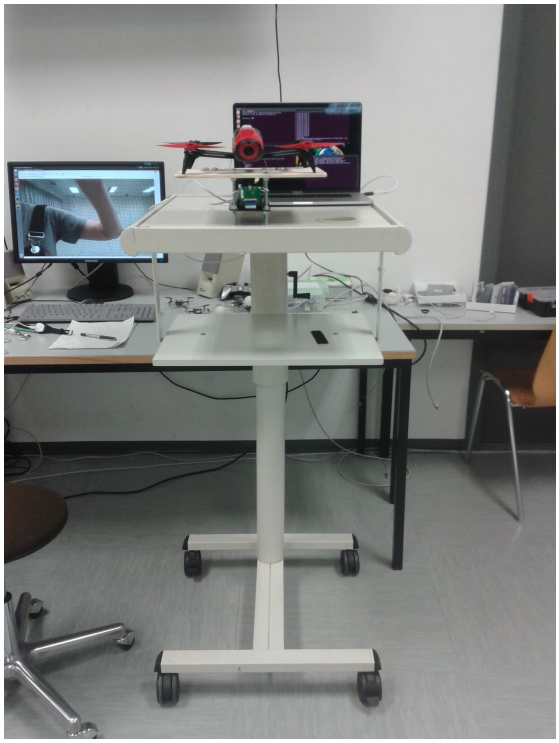

Figure 3.10. Mobile camera setup

When we reached the phase where we wished to acquire dynamic footage, we placed the structure on a wheeled cart (Fig. 3.10), which had adjustable height. This allowed us to collect data that is more similar to data collected while flying. The mobile setup served us well for several months. At some point we no longer needed footage from the Bebop, since we decided to focus on collecting Himax data, and the drone was simply detached from the structure. However, this setup proved to be cumbersome when we wanted to capture Himax images with different pitch angles. The desire to have precise control over the orientation of the Himax camera led to a new design - a case that would hold both the Gapuino and the camera board, and will allow us to attached it to a jointed arm (Fig. 3.11). The 3D model was written using OpenSCAD, and was printed in cooperation with the computational fabrication lab (Prof. Piotr Didyk). The case was designed to allow the placement of the IR filter on the camera lens, that would block the IR signal caused by the OptiTrack system. The .obj file (Fig. 3.11)for the Gapuino case will be shared upon request.

### 3.2.5 Data Visualization

Visualizing the datasets is a fundamental part of our development pipeline; to this end, we implemented various tools, described in the following. Their purpose is to analyse the correctness of the collected data, and ensure the collected data is diverse and covers a satisfying ranges of values and scenarios.

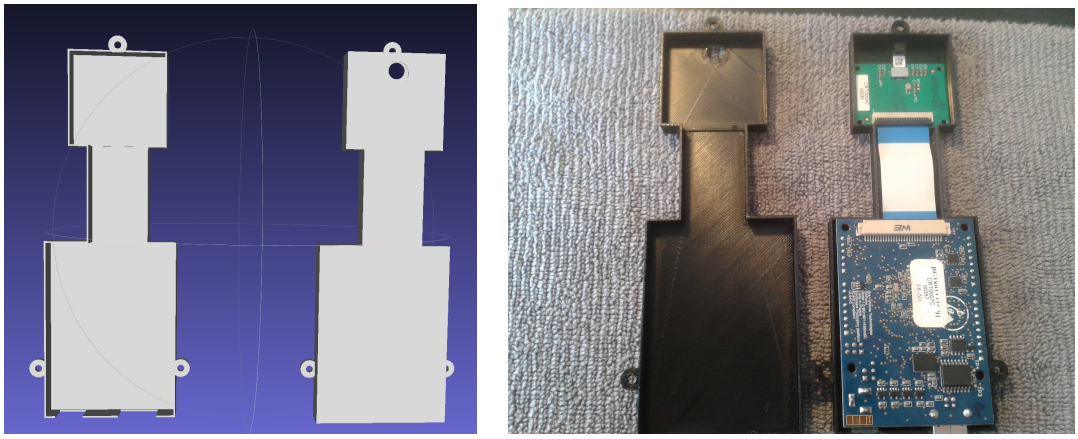

Figure 3.11. Left: 3D model of the Gapuino case. Right: The actual printed Gapuino case

### 2D World Frame Visualization

In order to verify the correctness of the acquired data, we implemented a tool that, given a dataset, generates a video where each frame contains:

- An image from the raw recorded data
- The GT values of both camera and head markers in the world frame
- An illustration of the Drone Arena, on which the markers locations are shown from top-view

A sample frame from the video is presented in Figure 3.12. The code is implemented in WorldTopViewGT.py. The grid in the plot corresponds to the grid painted on the Drone Arena floor, where each square is $1.2m \times 1.2m$. Plotting the position and orientation of both markers side-by-side with the video simplifies the identification of synchronization issues, in which a delay is introduced between the video frames and the labels.

### 2D Camera Frame Visualization

This visualization is similar to the previous and allows us to verify the correctness of the data, while examining it in the camera reference frame. Given a dataset, this tool generates a video where each frame contains:

- An image from the raw recorded data
- The GT values of both camera and head markers in the camera frame
- An illustration of the Drone Arena, on which the markers locations are shown from top-view

A sample frame from the video is presented in Figure 3.13. The code is implemented in CameraHeadGT.py.

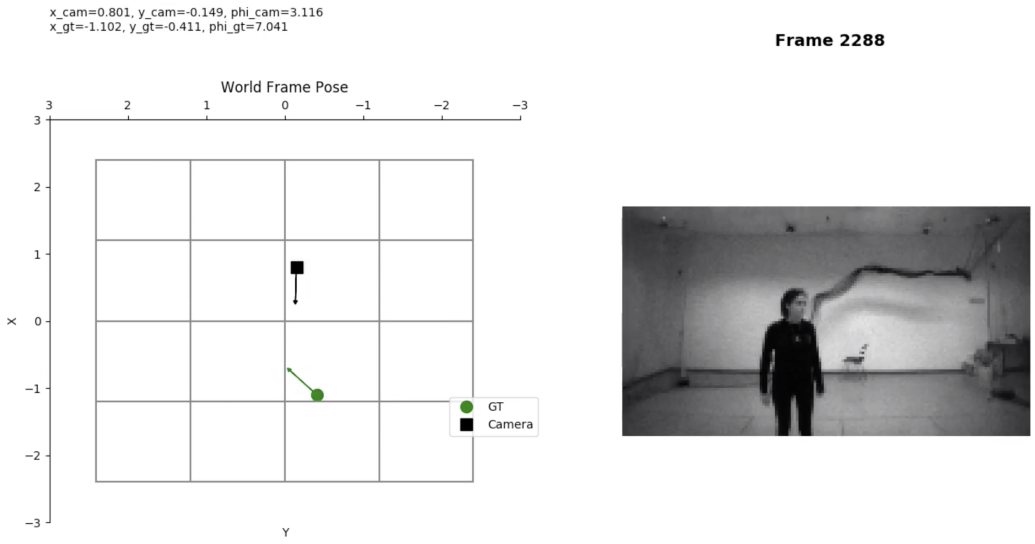


Figure 3.12. A frame from the 2D World Frame visualization video

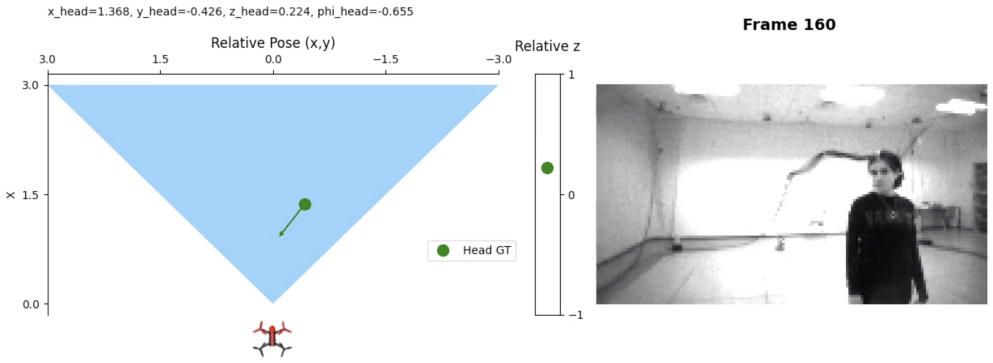


Figure 3.13. A frame from the 2D Camera Frame visualization video

#### Room Coverage Heatmap

It is important to have a good coverage of the room, in terms of our sample spatial distribution. We would like to know if some areas of the room, or orientations, are underrepresented and adjusted the dataset accordingly. For this purpose we wrote the script CoverageHeatMap.py, which produces a plot that presents the data as a heatmap. The plot (Fig. 3.14) is divided to 4 subplots, each corresponding to an orientation - N, W, S, E. Every heatmap represents how many samples were recorded in a certain area of space.

#### Range Histograms

In order to make sure our data is well-distributed in terms of the camera pose, we produce 4 subplots that displays the distribution of poses for each component - x, y, z and yaw. The poses are considered in the camera frame. This plot (Fig. 3.15) allows to deduce if the collected is diverse enough to cover the desired range of operation, and if it is well balanced. The code is implemented in ValuesHistogram.py.

### 3.2.6 Data Collection Format

The collected data was packaged using Pandas' DataFrame. The data structure has the following labeled columns:

- h - Frame height
- w - Frame width
- c - Number of channels
- x - Video frame data
- y - Relative pose between the drone and the object
- z - Drone position in the world frame
- t - Rosbag timestamp
- p - Drone pitch in the world frame
- r - Drone roll in the world frame (optional)
- o - Recorded prediction (optional)

Each row in the DataFrame represents a single video frame.

## 3.3 Metrics

The performance is evaluated in several ways:

- Mean square error (MSE) - measures the average square difference between prediction and ground truth values. Penalizes large errors more harshly than smaller ones.

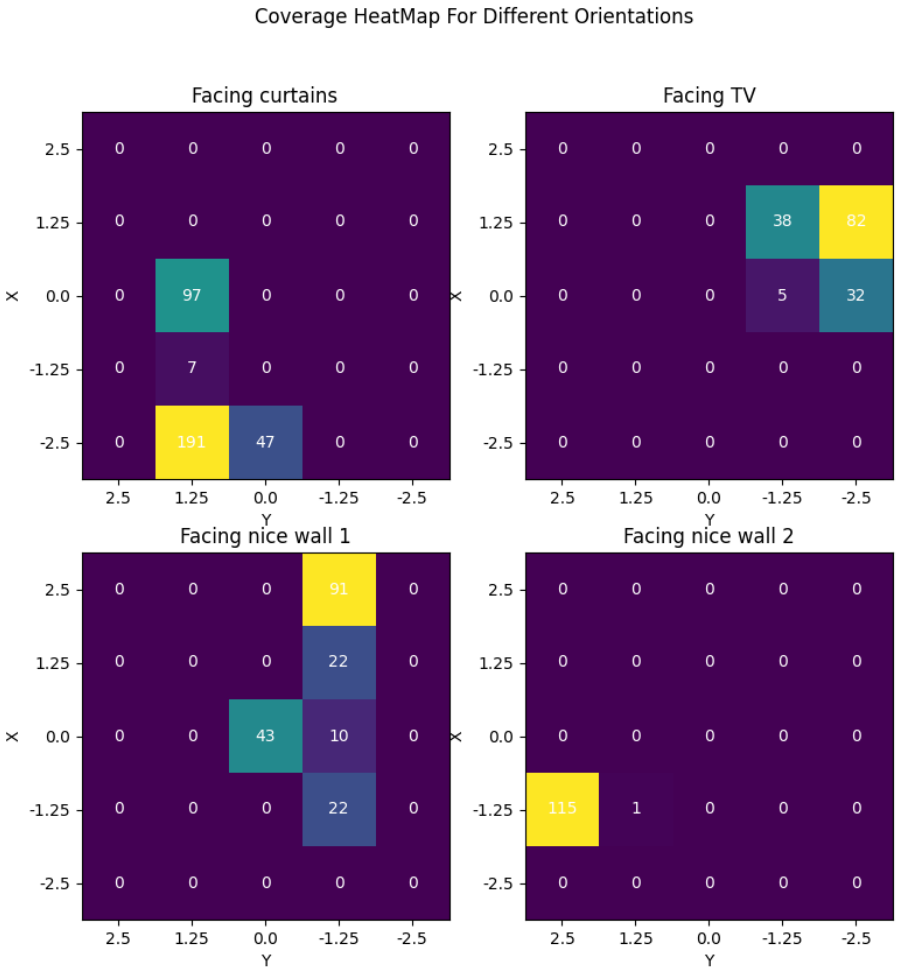


Figure 3.14. Coverage of the Drone Arena in a specific dataset

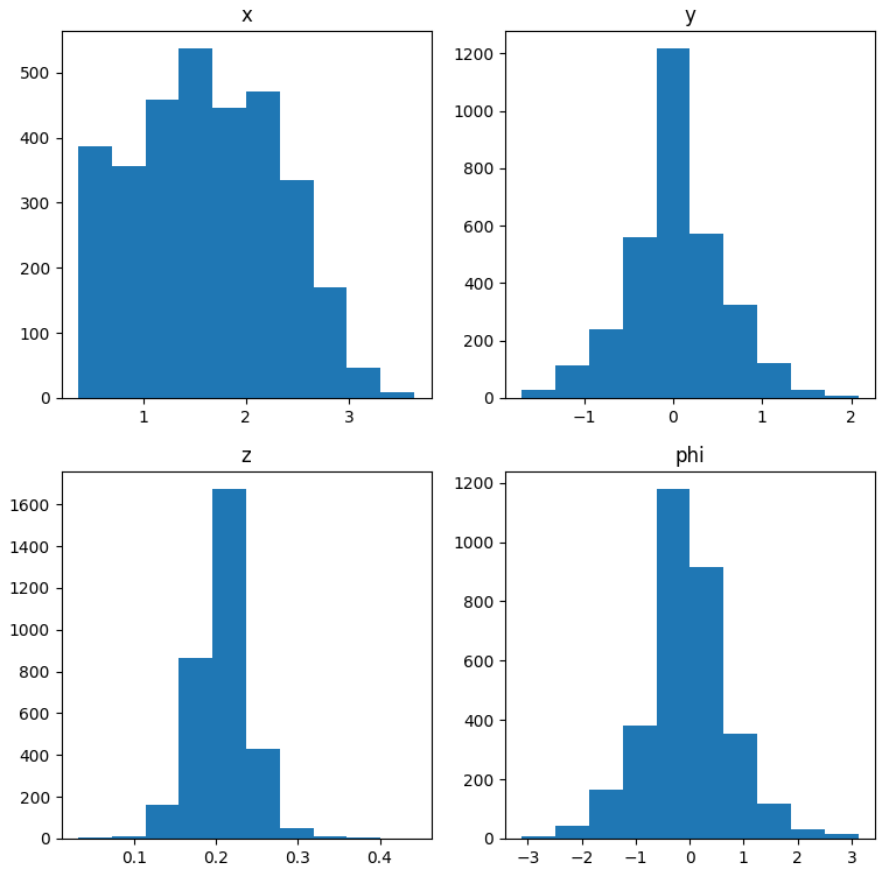


Figure 3.15. The distribution of collected poses, in the camera frame

- Mean absolute error(MAE) - measures the average absolute difference between prediction and ground truth values.

- $R^2$ score - measures to what extent the variance in the predictions explains the variance in the observed values.

The MSE and MAE reflects the accuracy of the model, but it is not trivial to determine which values would lead to a desired behavior. Even when a model produces moderate error in these metrics, its performance on the task might be reasonably good. We find that looking on the $R^2$ score is a better indicator of performance, which is why it is computed during the training and testing phases. Even for larger MSE and MAE values, if the $R^2$ score was above 0.6, the model performed fairly well in qualitative evaluations.

## 3.4 Training

Training of the model was done in PyTorch. ProximityNet was ported from Keras, and DroNet ported from TensorFlow. Everything that wrapped the NN, from data loading to performance analysis was re-written completely by us to fit the PyTorch API. In the work of [Mantegazza et al., 2019], two networks were examined - both take as input video frames, but the first outputs the pose variables $\langle x, y, z, yaw \rangle$ of the person relative to the drone, and the second one outputs the control variables of the drone (steering angle, speed). In the data collection process, each frame has the ground truth for the pose variables, provided by a MoCap system. Since these ground truth labels were readily available and easy to compare, we started by porting the first network.

The cost function that we used was L1 loss, which we adapted from ProximityNet. We use the Adam optimizer with a learning rate of 0.001, and a batch size of 64. Models were trained for 100 epochs.

Training was monitored by early stopping strategy, which ceased the training process once the validation score did not improve for 10 consecutive iterations. We collect information about the model's performance during the training process, which we visualize by the end of it. And example for how the learning curves look can be seen in Figure 3.16.

The data set was split 80:20 or 30:70 between training and test set. From the frames dedicated to training, 20% were used for validation and early stopping.

### 3.4.1 Data Augmentation

To promote better generalization we applied several augmentations during training. Some are general augmentation techniques, and some tailored to our problem specifically. A collage demonstrating the result of our various augmentations can be found in Figure 3.17.

#### Y-axis Mirroring

To ensure the model learns relevant features, and not correlations that are introduced by a subject's tendency to favor certain poses, every second image during training was flipped around the vertical axis.

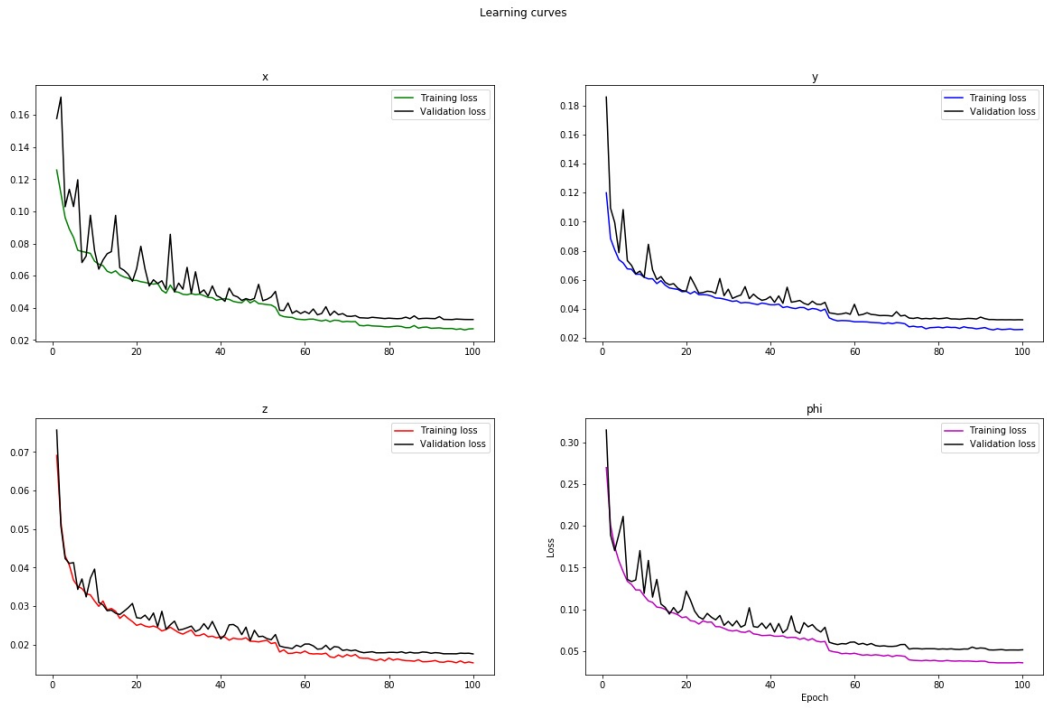


Figure 3.16. Evolution of the raining and validation loss during 100 training epochs

### Noise

Gaussian noise was added to the training data to increase the generalization power. Even between two Himax cameras there are variations in electronic, optic and mechanical properties that arise from an imperfect manufacturing process. The data collection was done using the Gapuino-boarded Himax camera, but the target is the pulp-shield with its own Himax camera. The assumption was that there were subtle differences between the cameras such as quantum efficiency. While shot noise is a Poisson process, for large number of photons, it is almost indistinguishable from Gaussian noise.

### Gamma correction

Gamma correction is a nonlinear operation used to manipulate the luminance values in an image. The operation does not change the range of the values, only the way tones are distributed. Augmenting the images by applying gamma correction allows to simulate scenes that are very bright or very dark, and also differences between cameras.

### Exposure

The exposure is the amount of light per area reaching a photoelectric cell. We altered an image's exposure by multiplying all pixel values by a factor, simulating more photons reaching the sensors. While it preserves the range, like Gamma correction, it is a linear operation.

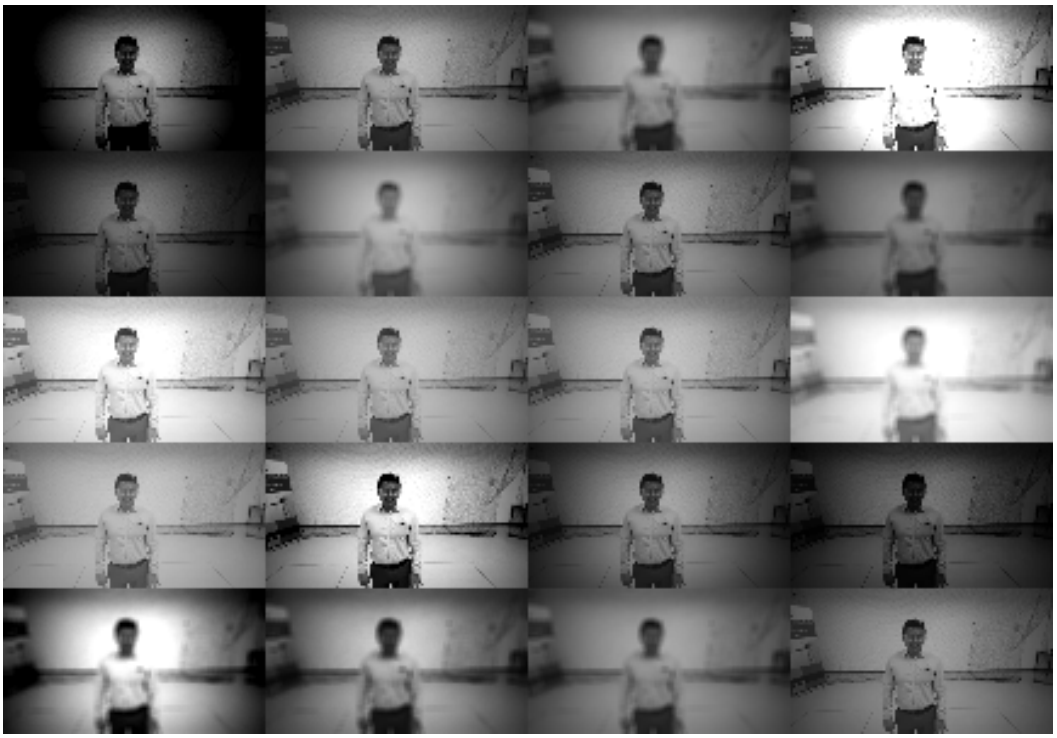

Figure 3.17. A single video frame after applying different augmentations

#### Dynamic range

Dynamic range in an camera is the ratio between the largest and smallest value of brightness it can capture. Augmenting the dynamic range means we are decreasing the amount of information in the image. Depending on the parameters we choose, we lose dark or bright details - tones that were distinguishable before will collapse into one, and some brightness levels at the edges will not be included.

#### Vignette

Vignette is an artifact caused by the mechanical properties of a camera, since the aperture is circular and the CCD is rectangular. Light is blocked by the aperture and does not reach the corners of the CCD, creating a reduction in the brightness around the periphery of the image. The vignette is extremely apparent in the Himax images, and vignette augmentation was added so that no unwanted correlation would be learned from this artifact.

#### Blur

Gaussian blur was added to images to encourage the model to learn features on larger scale, by suppressing small details. A model can learn to predict the yaw through fine facial details when in close range, but due to the poor quality of the camera, these details are lost when the subject is slightly further away. Also, since the number of participants in the dataset is not large, we want to avoid learning subject specific characteristics.

#### Pitch

Pitch is the angle applied to the camera around the x-axis. By changing this angle, but remaining in the same position, we are capturing images that contain different parts of the same scene. Depending on the angle, the image would include more of the floor or the ceiling. There is also a perspective transformation that changes the appearance of objects. Since the pitch of the camera can change dramatically during acceleration of the drone, it was important to augment images for pitch, so the pose can be estimated correctly for varying pitch values. Pitch augmentation is done by cropping the images vertically, which simulates the tilt of the camera.

#### Roll

Roll is the angle applied to the camera around the z-axis. Like with pitch, by changing this angle, but remaining in the same position, we are capturing images that contain different parts of the same scene. Roll augmentation is done by rotation transform on the images, which simulates the rotation of the camera.

## 3.5 Testing Methodology

Here we introduce the tools we developed in order to assess the performance of the model offline. Actual results will be discussed in Chapter 4 and Chapter 5, and this serves as an introduction. Purely quantitative measures such as MSE, MAE and $R^2$ score can give a general idea about the accuracy of the model, but they do not give a deeper insight about the weakness

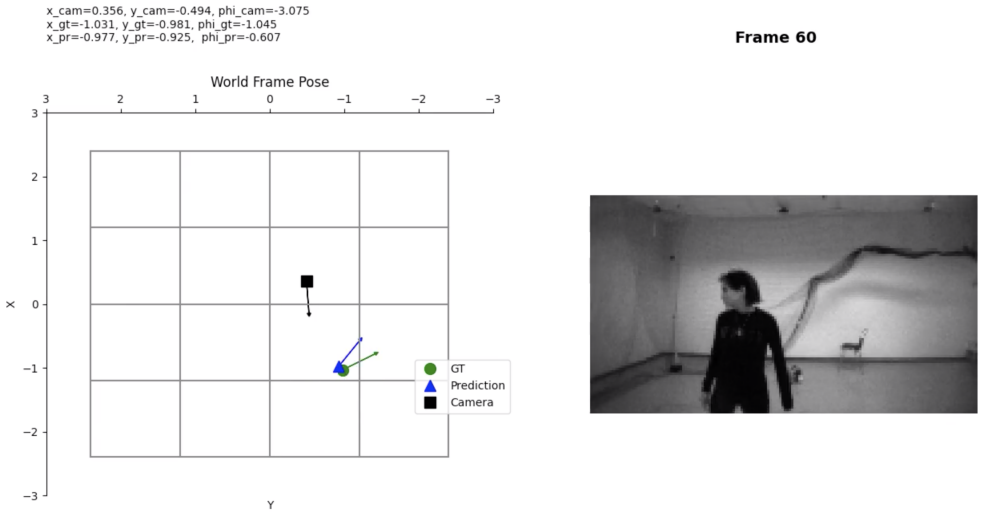


Figure 3.18. A frame from the WorldTopViewGTvsPred script video

of the network. We also noticed that while the standard measure might be mediocre, the on-board behavior is reasonable. Scripts that visualize the network predictions can paint a clearer picture than raw data, and allow us to speculate about the on-board performance.

### 2D World Frame visualization

An expansion of a similar script used for data verification, WorldTopViewGTvsPred.py produces a video, where each frame contains an image, the GT values of both camera and head markers in the world frame, the pose predicted by the NN in the world frame and a illustration of the DroneArena, on which the marker locations are shown from top-view. An example plot can be seen in Figure 3.18.

### 2D Camera Frame visualization

Similarly to the script used for verifying the correctness of the data, CameraHeadGTvsPred.py, produces a video, where each frame contains an image, the GT values of both camera and head markers in the camera frame, the pose predicted by the NN in the camera frame and a illustration of the Drone Arena,on which the marker locations are shown from top-view. An example plot can be seen in Figure 3.19.

### Error Heatmap

The script ErroreHeatMap.py produces 4 subplots. Each heatmap plot is corresponding to an orientation - N, W, S, E. Every heatmap represents how many samples were recorded in a certain area of space and what was the average prediction error. This allows us to see if in some areas of the room, or in some orientations, the learning process was sub-optimal and the NN shows degraded performance. An example can be seen in Figure 3.20.

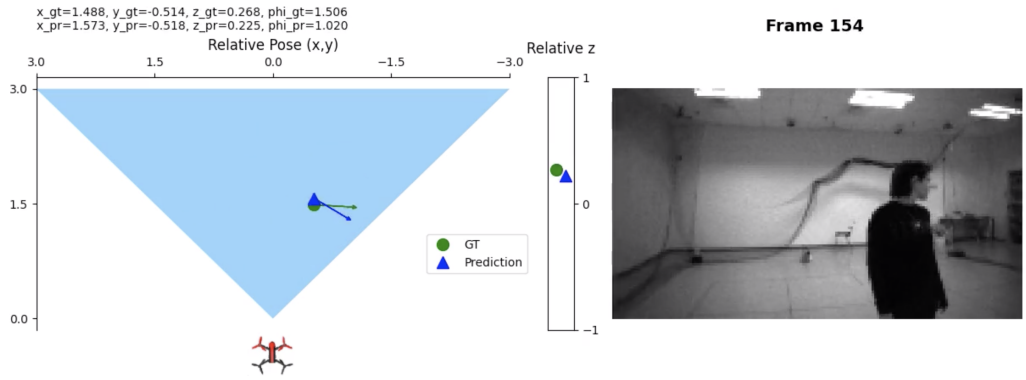


Figure 3.19. A frame from the CameraHeadGTvsPred script video

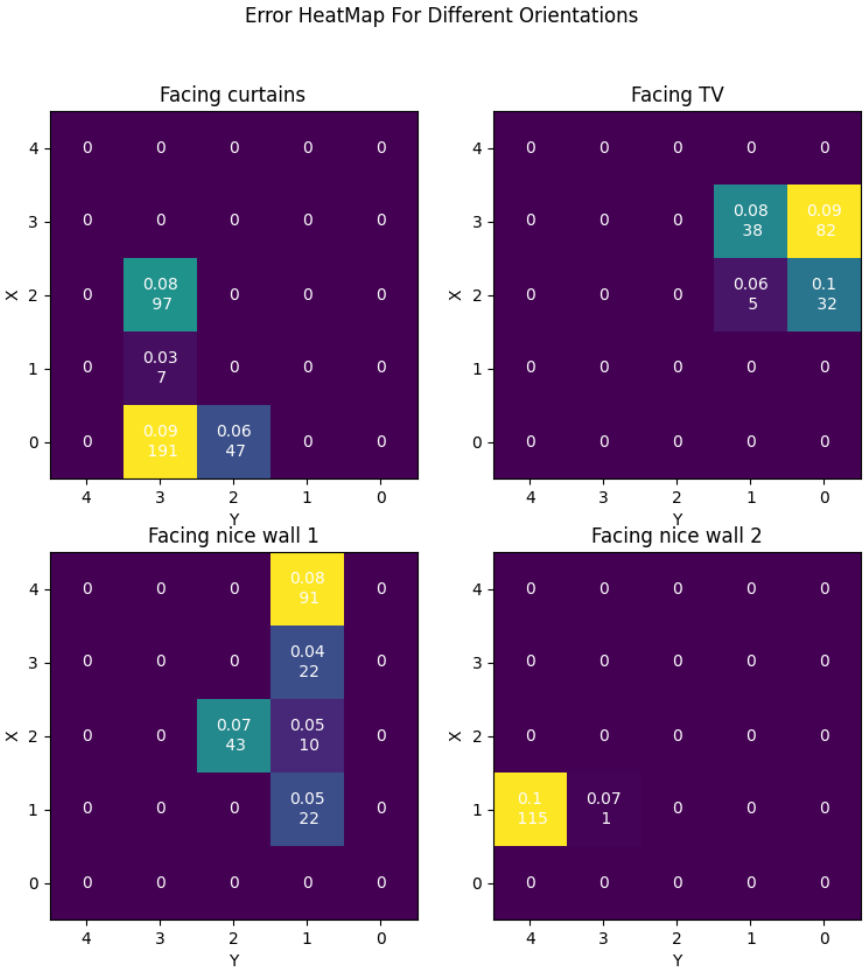


Figure 3.20. Error distribution in a specific dataset

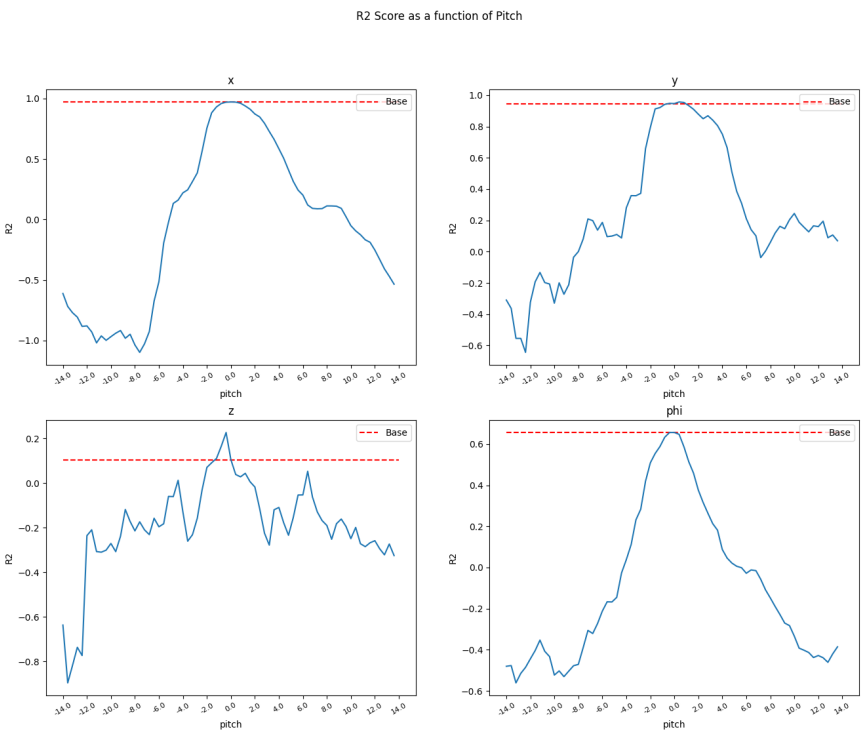


Figure 3.21. The impact of pitch changes on the $R^2$ score

### Pitch Impact

To estimate the sensitivity of our model to pitch variations, and to convince ourselves that the pitch augmentation are helping the model to generalize and adapt to pitch changes, we want to plot the prediction accuracy for different pitch values. To accomplish that, we wrote the script PitchvsR2Score.py, which produces 4 subplots. Each subplot corresponds to a component of the pose, and describes the effect of varying pitch on the prediction accuracy in term of $R^2$ score (Fig. 3.21). The varying pitch input can be real data recorded with different pitch angles, or simulated pitch created using augmentation.

## 3.6 Deployment

During the process of completing the thesis, two different deployment pipelines were used. The first pipeline was the same one used for the work of [Palossi et al., 2019a]. We experienced several failed attempts at deployment for the 160x96 models, and by that point, the old pipeline was abandoned by the developers in favor of the newer NEMO/DORY pipeline. We decided to continue with the new pipeline. While the implementation varies, both include 3 basic building blocks:

1. Quantization - performed by the NEMO library, which outputs a quantized version of

the model (weights and biases). For the first pipeline the quantization resulted in 16-bit elements, and in the second, 8-bit elements.

2. Neural network representation - C embedded code that describes the layer functionality. In the first pipeline structuring the neural network is done manually, with the help of the AutoTiler. In the second pipeline, the code generation is automated using DORY.

3. Application - a program that handles the image capture, inference, communication and startup and cleanup procedures. A different application was used for each pipeline, due to API changes, but the conceptually they were similar.

Aiming for embedded platform, led us to adjust the neural network hyper-parameters, like changing the original 3x3 pooling layer, to a more hardware-friendly 2x2 filters. Further changes made, such as switching architectures, and more hyper-parameter tuning, are described in length in Chapter 6.

### 3.6.1 PULP-App

The first pipeline used an application that was very similar to the one provided in the PULP-DroNet repository. Some additions were introduced, like downsampling images, and an option to stream images through a pipe into the host computer. In this application, image acquisition was triggered from the cluster, right after the first layer was computed, and the buffer dedicated for storing the image could be reused for capturing a new frame. The application was using the rt API and based on the PULP SDK [PULP] , and was deployed to the PULP-Shield [Mueller, 2018].

The applications implemented for the second pipeline (NEMO/DORY), was different in a several key aspects:

- API - NEMO/DORY was implemented using the pm-sis API, which required replacing all the functions in the original application with their pm-sis equivalent, and using the GAP SDK.

- Platform - we decided to target AI Deck instead of the PULP-Shield. we added UART functionality to communicate with the firmware, which replaced the previously used SPI.

- WiFi - as a result of moving to AI Deck, images can be streamed using the Nina WiFi module, even during flight.

- Advanced camera control - online correction to auto exposure parameters was introduced.

- Sequential image capture - to simplify the already complex integration, image capture was triggered sequentially after the inference ended.

### 3.6.2 Bug Hunting

During the verification process, we encountered interesting hardware bugs in the GAP8 chip. The first one was related to the DMA engine - when multiple calls for 3D data transfers were issued by different cores, the program hanged. It was solved software-wise inside DORY by the DORY developers. While trying to implement remote control that will allow us to change the camera parameters and switch on and off the WiFi module, we bumped into another bug. We

were unable to configure the HyperRAM and UART_RX to work simultaneously, although the GAP8 data-sheet revealed no reason for the conflict. However, as a result of another hardware bug, we had to toggle the pad configuration between the UART_RX and the HyperRAM to achieve the desired behavior in our app.

# Chapter 4

# Evaluation

In this chapter we examine the performance of a deployed model, recreating the demo described in [Mantegazza et al., 2019]. For the purpose of evaluation, we use the tools introduced in Chapter 3. The model that was chosen for deployment and final evaluation was a 160x96x32 PenguiNet model, where 160x96 refer to the size of the input image, and 32 is the number of channels in the first convolutional layer. The model was quantized and converted to c using the NEMO/DORY pipeline, with 8-bit quantization.

## 4.1 Dataset

The model was trained on a relatively small dataset that included images of me (Nicky) - less than 5000 frames. 4129 frames were dedicated to training, and underwent aggressive augmentation extending the training set to 41290 samples. 803 frames were saved for the test set. From the training set, 20% of the frames were dedicated for validation.

The reason for the limited data was the Corona-virus outbreak. We had very few videos including other people, and it was not clear if I would be able to test the model on others. So it was trained and tested on me (Nicky), with minor variations like hairstyle and clothes. This dataset does not include variations in the z axis, and for this reason the inference results are not considered in this evaluation. I (Nicky) collected this data set mostly alone, so a large part of the frames are static - only I move and the camera is static on the cart. I tried to add dynamic shots by pushing and pulling the cart to create camera motion.

A heatmap of the camera poses in the data-set can be seen in Figures 4.1 and 4.2. Sample frames from the raw training set, the augmented training set and the test set are presented in Figures 4.3, 4.4 and 4.5.

## 4.2 Controller

The output of our model is $(x, y, z, \phi)$, the relative pose of the user with regard to the drone. A controller must be implemented in order to track the user, based on the prediction data. The controller we used is based on the controller in Mantegazza et al. [2019], and the code can be found in here . The control flow is as follows:

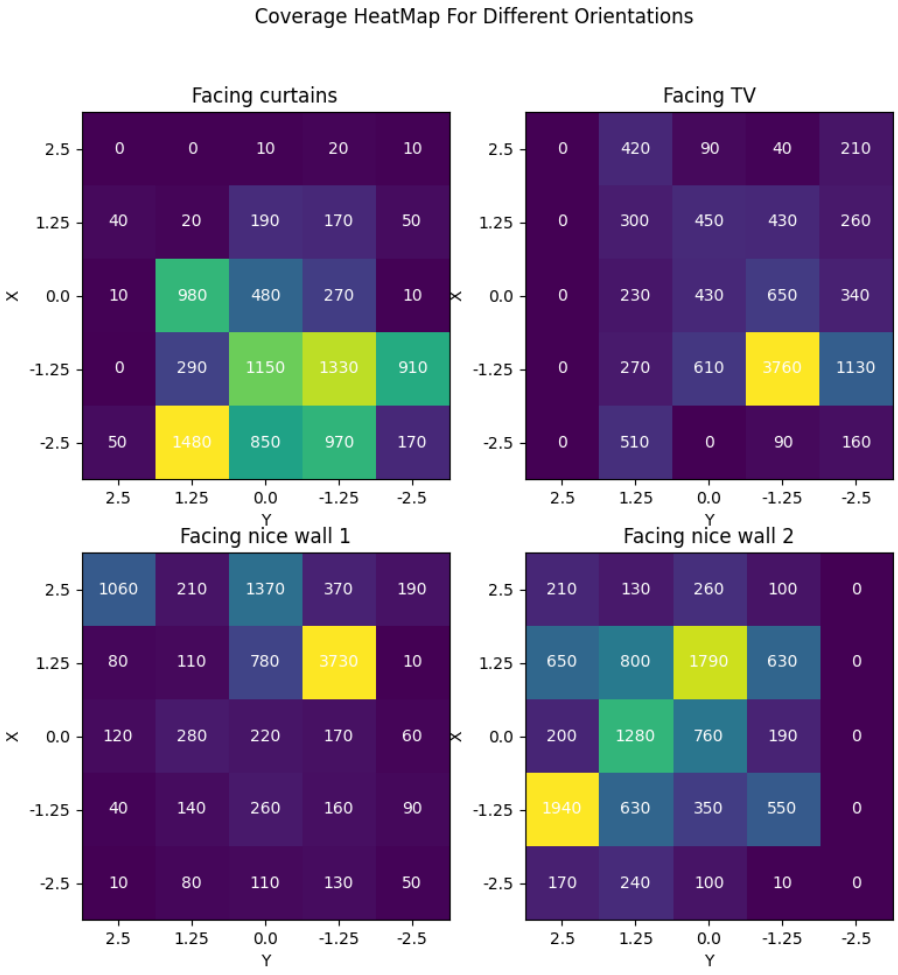


Figure 4.1. Room coverage of the samples in the train set of Nicky dataset.

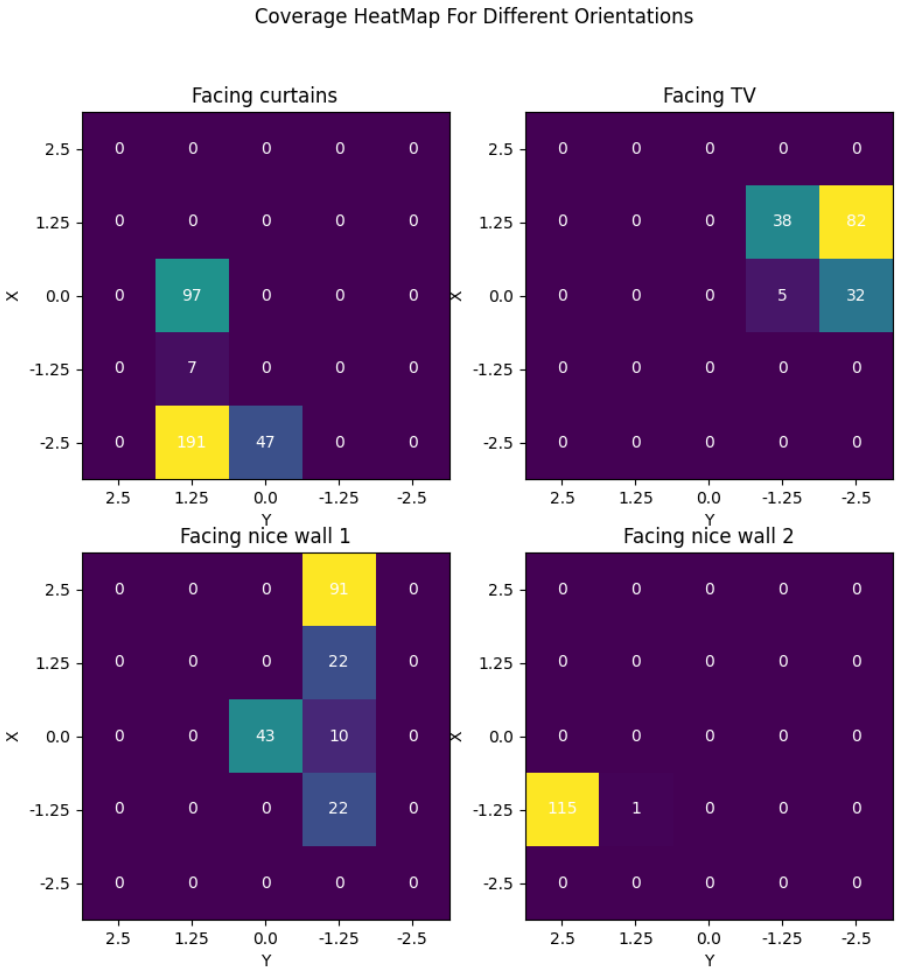


Figure 4.2. Room coverage of the samples in the test set of Nicky dataset.

Figure 4.3. Sample frames from the raw train set of Nicky dataset.

Figure 4.4. Sample frames from the augmented train set of Nicky dataset.

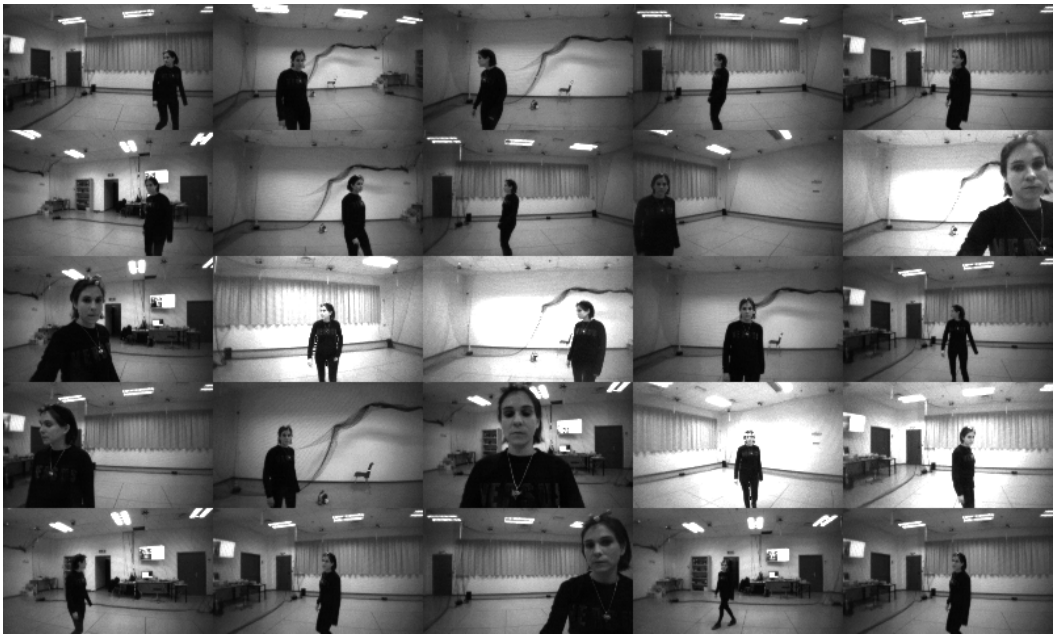

Figure 4.5. Sample frames from the train set of Nicky dataset.

1. Calculate the target pose by adding up the predicted pose of the user, and the distance we would like to keep from it.

$$X_{\text{target}} = X_{\text{pred}} + d_{\text{desired}} \tag{4.1}$$

   We set $d_{\text{desired}} = 1.3m$

2. Compute the target velocity by considering the current pose, target pose, and accounting for the user velocity

$$V_{\text{target}} = \frac{(X_{\text{target}} - X_{\text{curr}})}{\eta} + V_{\text{user}} \tag{4.2}$$

   With $\eta = 1$ in our experiment. The head velocity is predicted using a Kalman filter.

During both stages, there are various safety checks taking place, such as the virtual fence, and maximal speed limitation (1 m/s, in our case). Since the Crazyflie has smarter on board controls, the only information they require is the desired pose and velocity. The flow in the original Bebop controller was more sophisticated, computing the desired acceleration and attitude, since these were the required inputs for the drone control.

## 4.3 Off-Board Evaluation

### 4.3.1 Quantitative Evaluation

It was important to show that the deployed model and the infrastructure required for inference is accurately reproducing the behavior of the original PyTorch code. For that we recorded the

| $R^2$ Score | | |
|---|---|---|
| $x$ | $y$ | $\phi$ |
| 0.999 | 0.999 | 0.997 |

Table 4.1. $R^2$ scores describing the correlation between the full precision and the quantized predictions.

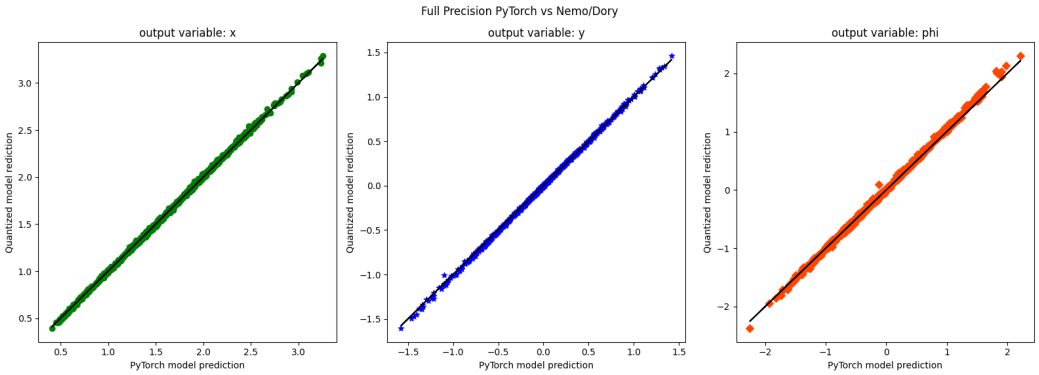


Figure 4.6. The embedded, quantized model predictions as a function of the PyTorch predictions for a PenguiNet model

outputs of the full precision PyTorch model on a test dataset. We used the gvsoc, the GAP8 emulator, to run inference on the same test set, and collected the predictions. The NEMO quantization aims to be numerically exact, and the DORY code is expected to correctly convert the PyTorch implementation to ANSI c. Thus, we expected the predictions of the full precision PyTorch model, and the embedded, quantized model running on the gvsoc, to be highly correlated. Ideally, the process of deployment will turn out to be an identity mapping. This objective was achieved as can be seen by Table 4.1 and Figure 4.6.

### 4.3.2 Qualitative Evaluation

Quantitative evaluation can be a bit too "dry", or not tell the entire story. Online evaluations are action-packed and it is hard to pay attention to the fine details of the task execution. This motivates our off-board qualitative evaluation, where we can "freeze" a frame in time and analyze the scene and the behavior of our model, with visual aid. For the qualitative evaluation we utilized our visualization scripts, to show the prediction and their visualization in the drone frame, side-by side with the video. We examine the full precision PyTorch prediction against the quantized, embedded predictions as we did in the previous section, but we also consider how they relate to the actual relative pose of the user. We expected to see correlation between the movement of the user in the video, and the plot describing the user's pose in the drone frame based on the outputs.

As can be seen in Figure 4.7, both models gave reasonable predictions that aligned with the video inputs. The predictions identified yaw variations, correctly assessed the user positioned on the edge of the camera's field of view, and gave a valid estimation of the user's relative distance. Verifying the predictions appear reasonable in this qualitative assessment was a pre-

| $R^2$ Score | | |
|---|---|---|
| $x$ | $y$ | $\phi$ |
| 0.989 | 0.975 | 0.955 |

Table 4.2. $R^2$ scores describing the correlation between the full precision and the quantized predictions.

liminary step to the next phase - online evaluation.

## 4.4 On-Board Evaluation

### 4.4.1 Qualitative Evaluation

To simplify the implementation, the application code was performing the frame capture and the inference sequentially. This is in contrast to the origin application of PULP-DroNet, where the image acquisition was triggered after computing the result of the first layer. Despite the sequential handling, the inference occurred at 10+ FPS, resulting in a desired reactive behavior of the nano-drone. The nano-drone tracking the user can be seen in this video , and the task as captured from the drone's point of view is documented here . Even though the dataset used for training the deployed model consisted only images of me (Nicky), it behaved surprisingly well when tracking other people. It managed to track me (Nicky) for the vast majority of time, and very sharp and fast movements were required to "shake it off". The drone behaves better when the background is not cluttered, so the performance was slightly decreased for the Drone Arena wall which contains lots of equipment.

### 4.4.2 Quantitative Evaluation

To get a more accurate estimation of quality, we created a setup that will allow us to record key measurements:

- Images were streamed from the AI Deck, using the Nina WiFi module, and published using ROS
- A ROS node which was subscribed to these image messages, computed the inference using the full precision PyTorch model, and published the results.
- The inference on-board was also published
- Markers were placed both on the user and on the AI Deck, and their positions were tracked using the OptiTrack system.

As before, we wanted to see that the inference on-board, resulting from the quantized, embedded model faithfully represents the full precision PyTorch model. The correlation was indeed very high, as can be seen in Table 4.2.

Since the deployment seemed to be successful, we turned to estimate how well we performed on the task. The online inference results were considered against the ground truth from the OptiTrack, displaying reasonable correlation (Table 4.3 and Figure 4.8). The ground

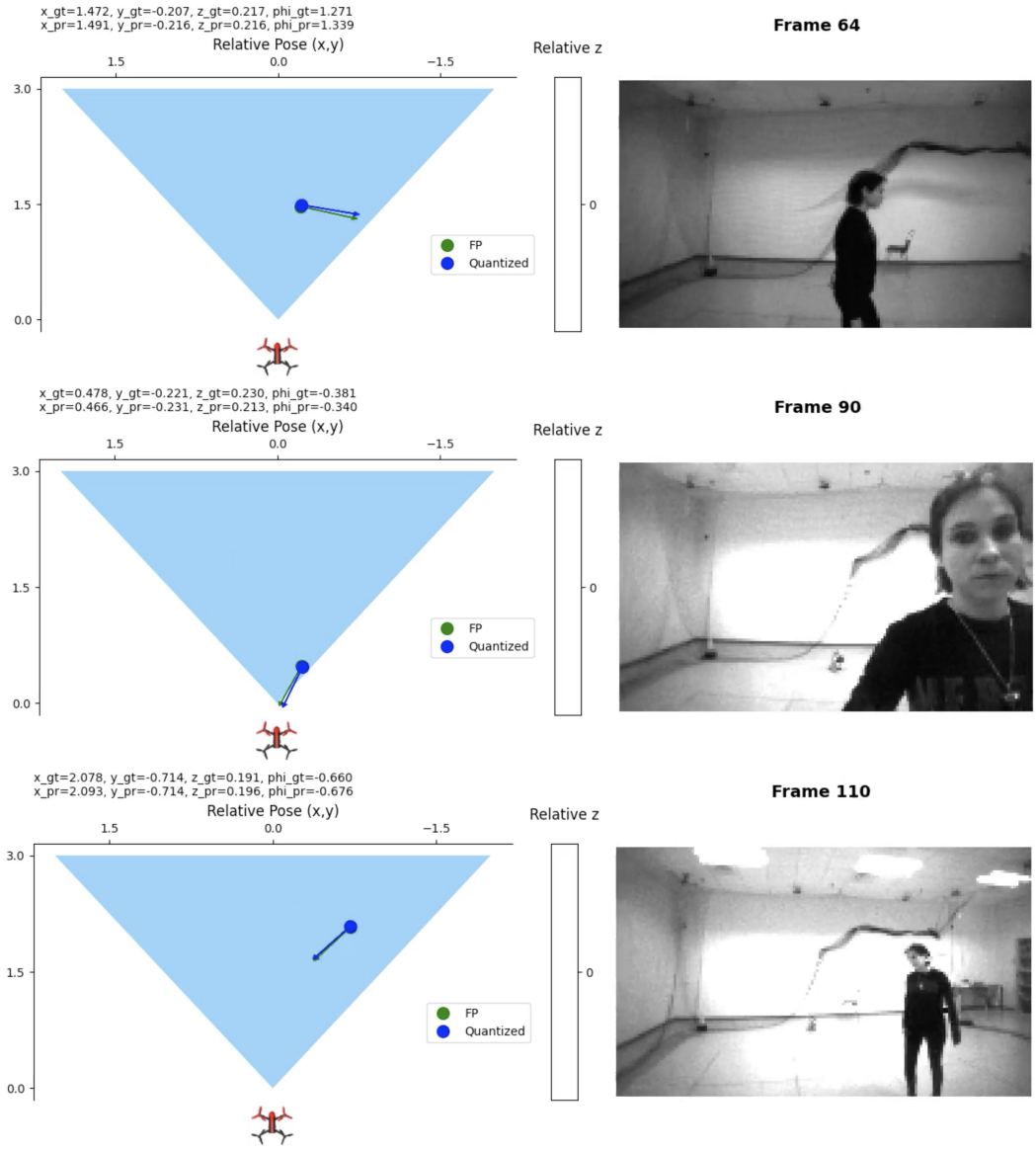


Figure 4.7. Examples of frames from the video used for qualitative estimation

| $R^2$ Score | | |
|---|---|---|
| $x$ | $y$ | $\phi$ |
| 0.871 | 0.723 | 0.633 |

Table 4.3. $R^2$ scores for the online inference results and the ground truth values from the OptiTrack.

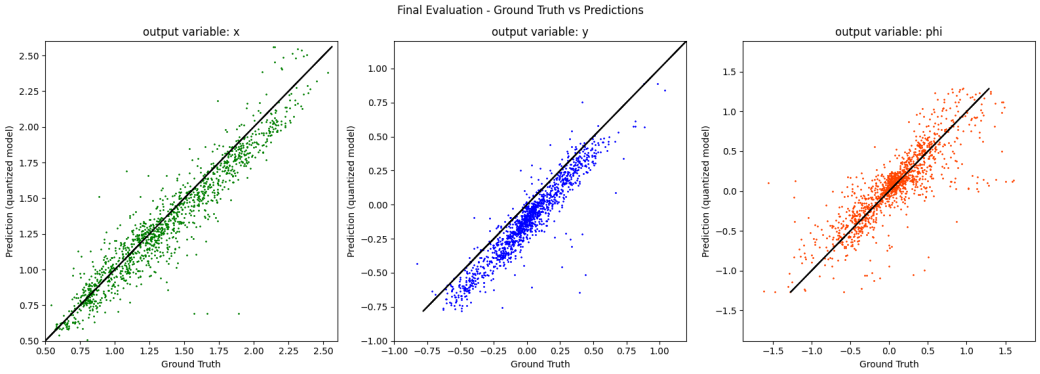


Figure 4.8. Predictions from the quantized model plotted as a function of OptiTrack poses for 1200 frames recorded during the final evaluation.

truth from the OptiTrack was also plotted against the quantized, on-board predictions in Figure 4.9.

The markers attached to the AI Deck were pushing its payload limits, so it was a little bit shaky during the flight. Because the user was moving vigorously trying to challenge the nano-drone, the headband with the marker slipped slightly to the side (figure 4.10), introducing a bias to the poses tracked by OptiTrack. Fortunately, the bias was constant and it was easy to detect, compute and remove. It appears that there is one data point, at around frame number 1200, where the OptiTrack system failed to track the user pose correctly. The other, less probable, option is the the user teleported 3m along the y axis. This sample was not removed from the recording data as we wanted to present raw, unprocessed results. Another aspect of verifying our correct implementation was in analyzing the residual error, the difference between the ground truth provided by the OptiTrack system, and the on-board predictions. Assuming our data was well-distributed, and the embedded implementation was correct, we expected to see a symmetric distribution of errors. Our expectations were mostly realized, as can be seen in Figure 4.11. The single faulty OptiTrack localization discussed above is not included in the plotted range in order to visualize symmetry clearly.

In addition to evaluating the accuracy, we also measured the performance in terms of time and power. The application ran on the hardware, connected to to a power measurement system. The GAP development board has several test points that can be used to directly measured the power consumption of the board. These test points are also present in the cluster and the fabric controller, which are of interest to us. While the program executes continuously, a pair of test points are measured using a voltmeter. Since the resistor connecting pairs of test points is 1Ω, the measured voltage will equal the consumed current, I. Then to measure the voltage at a test point, V, one of the voltmeter probes should be connected to ground (GND). The power consumption is computed as

$$P = I \times V \tag{4.3}$$

For more detailed measurement, and oscilloscope can be used. The fabric controller and cluster frequencies were set to 100MHz, and the VDD=1.0V. The most intense part off the application is the inference, which requires on average 32mW, with peak power consumption of 44mW, for both cluster and the fabric controller. The cluster is the main power consumer, which is

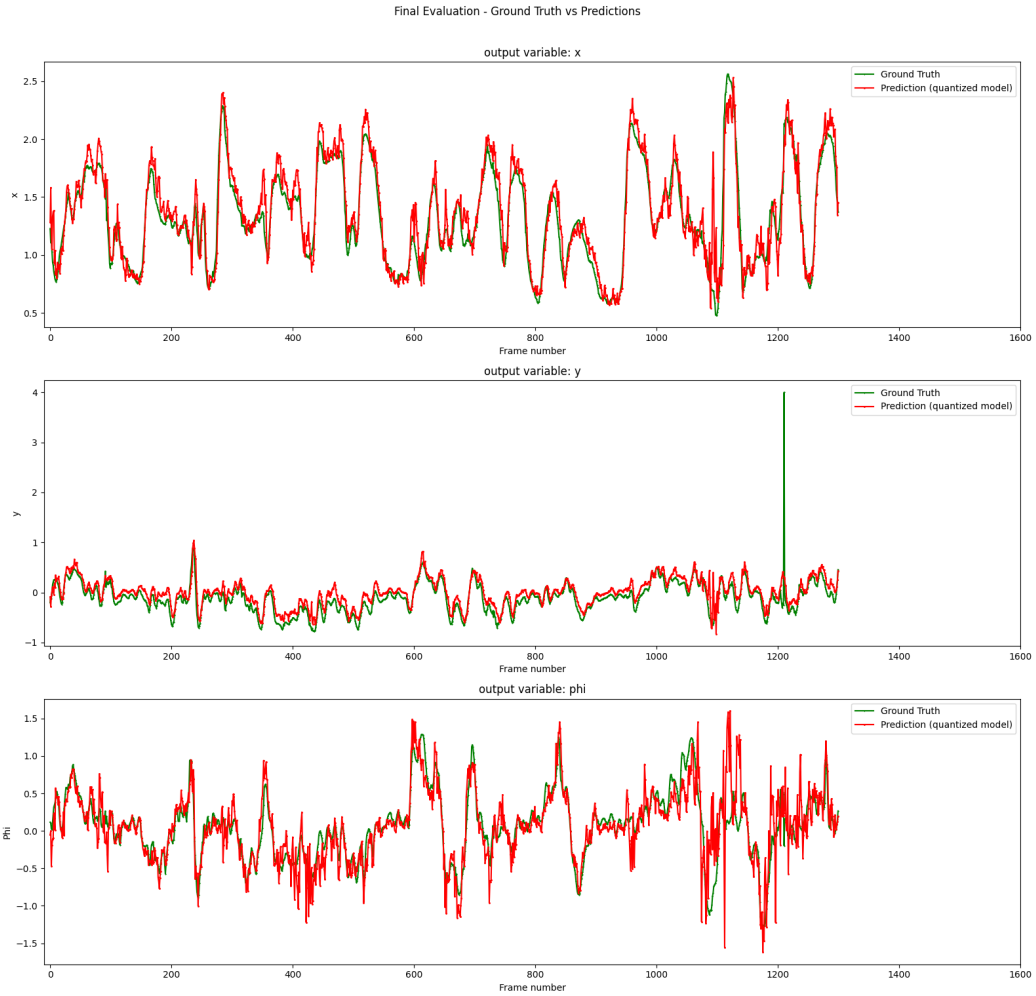


Figure 4.9. OptiTrack poses, and predictions from the quantized model plotted for 1200 frames recorded during the final evaluation.

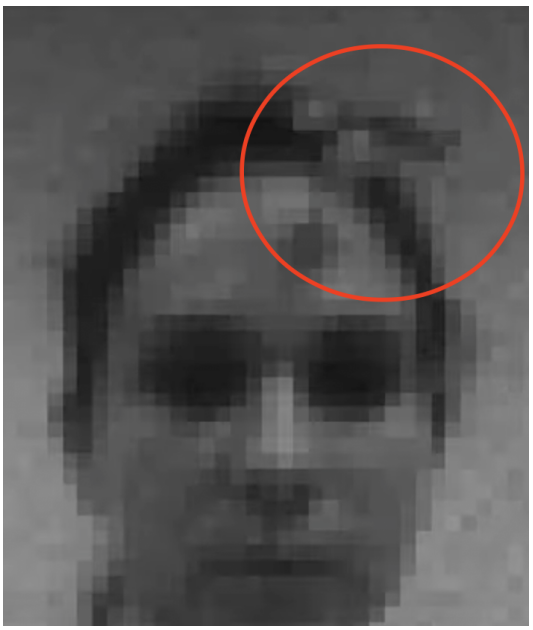

Figure 4.10. A frame captured by the Himax camera showing the marker has slid to the side of the user's face, where it should have been centered at the top of their head.

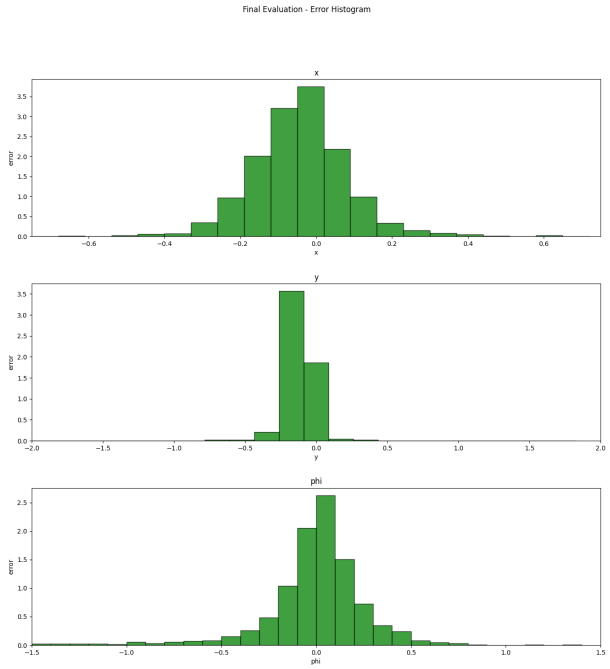


Figure 4.11. Histograms of the error along the different pose components

expected from an accelerator that is well-utilized by an application. It took us 70ms to complete an iteration, which is 14 FPS. The measured results are presented in Figure 4.12.

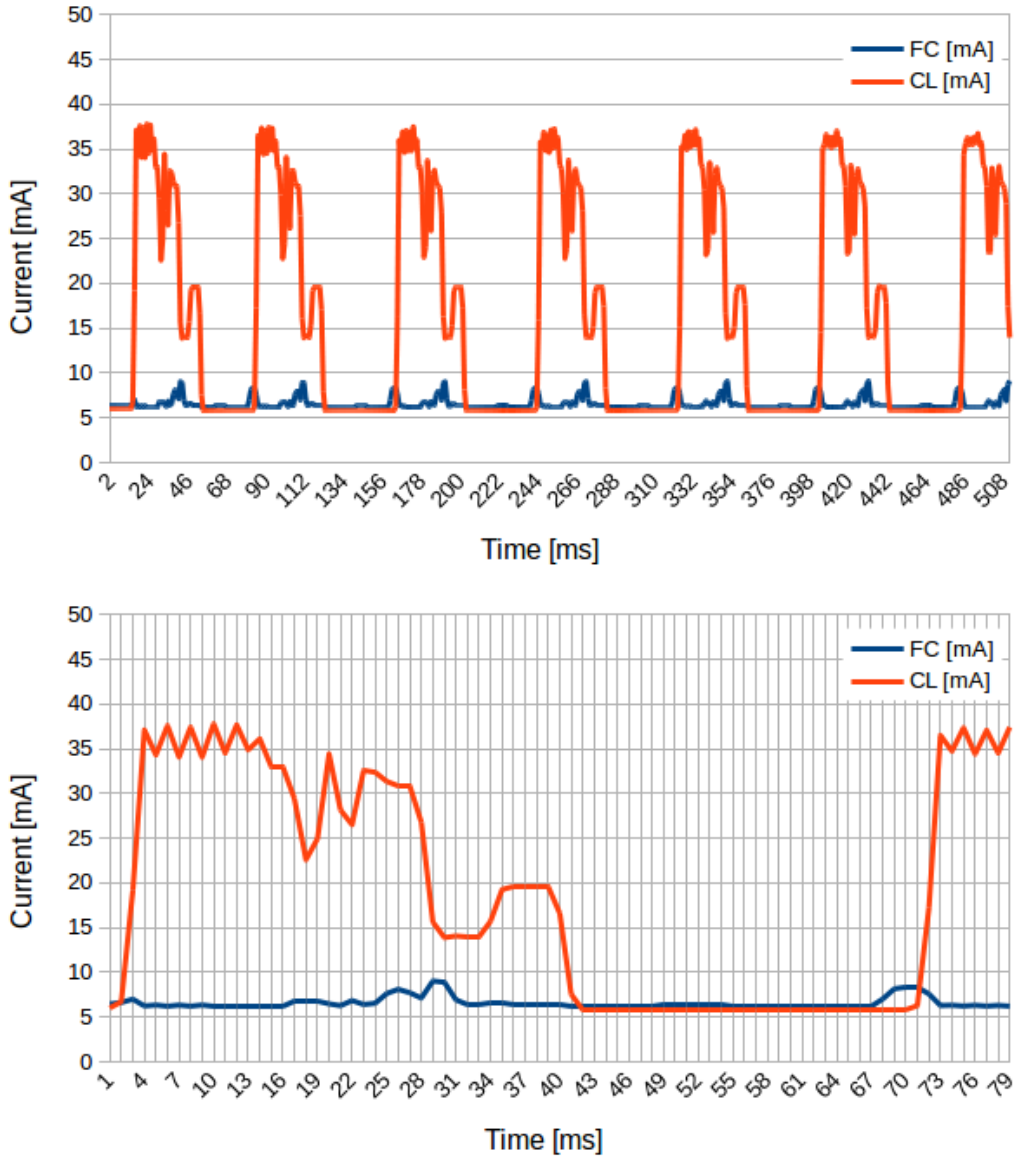


Figure 4.12. Power consumption and time analysis for the application. FC stands for fabric controller, and CL for the cluster. Top: Multiple iterations. Bottom: Zoom in on a single iteration.

# Chapter 5

# Exploration

The constraints of having a very minimal hardware led us to explore various ways to overcome obstacles associated with the low computational capabilities and the low fidelity sensors. In general we concern ourselves with axes of quality:

- Model Accuracy
- Performance (fps)
- Energy consumption

It is a challenge to excel even in two out of the three, as pulling in one direction affects the others. There is a trade-off between these three factors, and the best combination of accuracy, performance and energy efficiency is entirely dependent on the task at hand. In this chapter we explore techniques to improve these qualities, and also discuss how boosting a certain factor impacts the others. When referring to models in this section, we use the notation WxH or WxHxC, where WxH is the size of the input image, and C is the number of channels in the first convolutional layer.

## 5.1 Dataset

For the exploration part a dataset called "Others" was used. It is also quite small, less than 4000 frames, but includes footage of 6 people, with and without face masks. A heatmap of the camera poses in the data-set can be seen in Figures 5.1 and 5.2. Sample frames from the raw training set, the augmented training set and the test set are presented in Figures 5.3, 5.4 and 5.5.

This dataset is harder than the Nicky dataset, since it requires the model to generalize better. In Chapter 4 we chose the Nicky dataset (Fig. 4.3), for the purpose of demonstrating a working demo on a single person. But since this chapter is more exploratory, we thought it was fitting to choose a more challenging, diverse dataset.

2629 frames were dedicated to training, and underwent aggressive augmentation extending the training set to 26290 samples. 1119 frames were saved for the test set. From the training set, 20% of the frames were dedicated for validation.

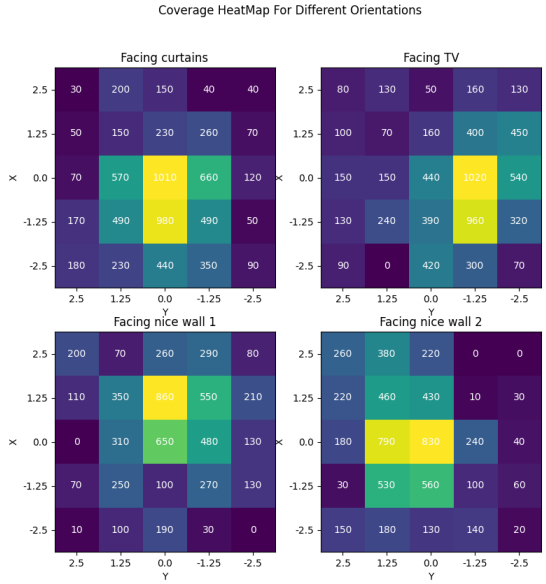


Figure 5.1. Number of images in the training set for the "Others" dataset, available for different camera positions (axes of subplots) and orientations (different subplots). The numbers are multiple of 10 due to the data augmentation.

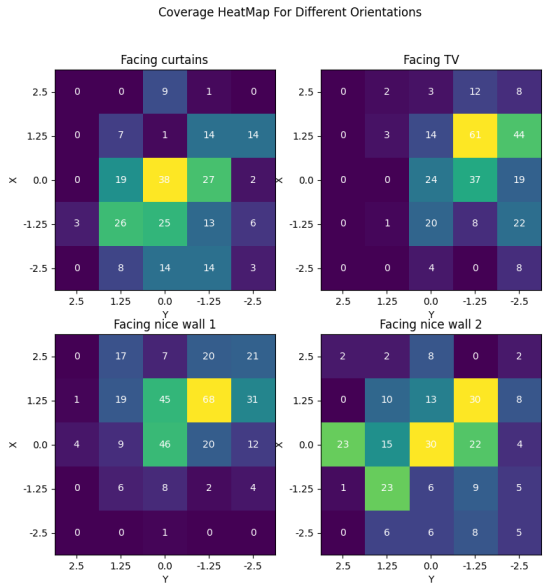


Figure 5.2. Room coverage of the samples in the test set of Others dataset.

Figure 5.3. Sample frames from the raw train set of Others dataset.

Figure 5.4. Sample frames from the augmented train set of Others dataset.

Figure 5.5. Sample frames from the test set of Others dataset.

| Visual Aug. | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| No | 0.23 | 0.19 | 0.08 | 0.53 | 0.76 | 0.78 | 0.54 | 0.13 |
| Yes | 0.215 | 0.2 | 0.07 | 0.45 | 0.79 | 0.72 | 0.61 | 0.3 |

Table 5.1. Test score for DroNet architecture with input of 160x96 for visually augmented model and non augmented model. Test set is raw Himax frames.

## 5.2 Visual Augmentations

The dataset we collected was very small ( 2600 frames). It was clear that in order to produce a working demo we would need to extend it. However, even if we were able to acquire a larger dataset, there are still reasons to support visual augmentations.

As previously discussed, the Himax camera produced images with a visible vignette. Additionally, the behavior of the auto exposure was hectic, going between incredibly dark frames to very bright ones within a fraction of a second. Also, considering it is a low-end camera, we expect to see some variance between different cameras of the same model, due to manufacturing artefacts. These reasons motivated aggressive visual augmentations in hope that our model would learn to cope with the input, even under significant deviations.

We trained a model on raw photos from the Himax without augmentation, and another model on the same training set but with heavy augmentations. We tested the models on two test-sets, raw-footage and an augmented version, and we noticed a dramatic increase in performance for the augmented model on all benchmarks, especially in predicting the yaw (Table 5.1). We also tried both models online on the Crazyflie, and qualitatively, the behavior of the augmented model was preferable.

| Dataset | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| Bebop | 0.71 | 0.37 | 0.28 | 0.57 | -0.99 | 0.39 | -1.76 | 0.06 |
| Himax | 0.21 | 0.17 | 0.08 | 0.43 | 0.79 | 0.79 | 0.59 | 0.39 |

Table 5.2. Test score for DroNet architecture trained on different dataset, and tested on Himax images

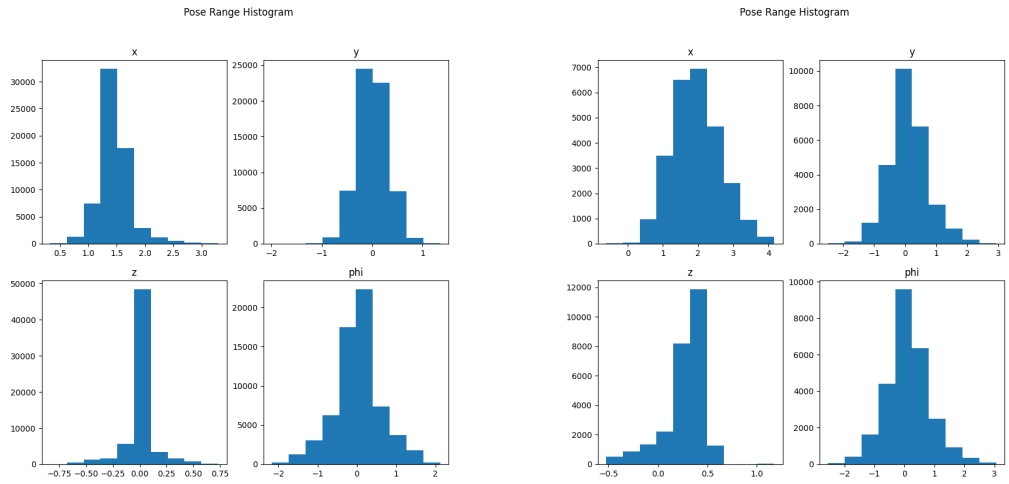


Figure 5.6. Left: Histogram of the pose components for the Bebop dataset. Right: Histogram of the pose components for the Himax dataset

## 5.3 Himax vs Bebop

Initially we were hoping to reuse the data collected by [Mantegazza et al., 2019] to train a model that would perform well given input from the Himax. As the cameras are at two different ends of the quality spectrum, we came up with a degradation process that would transform the Bebop images to a Himax equivalent. Despite the success we had with visually augmenting Himax images, applying the same tools on Bebop images was not quite effective. Training on transformed bebop images resulted in poor performance on Himax images, both offline (Table 5.2) and online. There is a possibility that the fault lies not in the degradation process, but in the limited range and deviation covered by Dario's dataset. The data was collected to train an end-to-end solution, where controls were directly infer from the input image. For this purpose the drone was controlled by an experienced pilot that mimicked the desired behavior of keeping a certain distance from the user and facing him. Thus, the x range of the relative pose is quite limited, as well as the yaw values. As can be seen in Figure 5.6, the Himax datasets include a wider range of poses, it might result in poor performance on the bebop-trained model.

## 5.4 Pitch Augmentations

The affect of pitch on prediction accuracy is something we noticed while testing the model online. Even though offline, the prediction of the x component was reasonable, when deploying the model on the Crazyflie, there was an obvious decrease in performance. We ran the model on the Bebop, applying a degradation process that made the images captured more similar to the Himax ones, and we could not reproduce the same faulty behaviour. A more in-depth investigation revealed that the Bebop camera had a video stabilizer that compensated for the pitch caused by accelerating and decelerating.

All the training data recorded with the Bebop therefore had pitch=0, and the images captured with the Himax were taken using the cart, where the camera was also facing forward without a pitch. As the PULP-Shield does not have a video stabilizer, the model was exposed to images with different pitch only during online testing. This explains the accuracy loss on the x axis, where the accelerations are most prominent and result in dramatic pitching of the drone.

To overcome this problem, we decided to collect data with varying pitch angle. We encountered 3 main constraints:

1. Crazyflie flight recording - The Crazyflie does not have enough storage memory for data collection. Streaming the frames to a PC via WiFi or other protocols was not supported at the time. The only possibility was flying the Crazyflie while it is tethered - there was a concern that the cable can drastically influence the drone dynamics, and potentially lead to accidents wee could not risk, as we had only one working drone.

2. Cart recording - As the cart structure was improvised, we had no appropriate equipment to control the camera's orientation. Tilting it by hand while driving the cart proved to be a test of coordination and dexterity none of us could pass.

3. Bebop flight recording - We managed to disable the video stabilizer on the Bebop, which allowed us to record videos with varying pitch. However, trying to transform Bebop images to Himax equivalent was unsuccessful, as discussed in the dedicated section.

Since capturing real data with different pitch angles seemed impossible, we were faced with a dilemma - should we ignore the model's pitch sensitivity, or try to overcome it by simulating pitch. We decided to fake the pitch by simply cropping the image differently (Fig. 5.8). The image captured by the Himax was 160x160, and the input to the model was 160x96, so we had some freedom on the vertical axis, cropping the image starting from different rows. A 64 pixel vertical distance between a frame cropped at the top, and a frame cropped from the very bottom translates roughly to a pitch range of $[-14°, 14°]$. The vertical FOV of the Himax is 65.65° , spanning across 160 pixels. To convert pixels to degrees and vice versa we used:

$$1\text{px} = \frac{\text{vfov}}{h} = \frac{65.65}{160} = 0.41° \tag{5.1}$$

A new model was trained with pitch augmented data, where each frame of the original dataset was processed to produce 20 different variation of pitch and visual augmentations. Comparing this model against the previously trained models, we noted a small improvement on the original test set with pitch 0. We created a new test set from the original one, where each frame is cropped vertically in 64 times. The visually augmented model performed poorly on it, but the model that included pitch augmentations performed well, as can be seen in Table 5.4 and figure 5.7.

| Pitch Aug. | Visual Aug. | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| No | No | 0.23 | 0.19 | 0.08 | 0.53 | 0.76 | 0.78 | 0.54 | 0.13 |
| No | Yes | 0.215 | 0.2 | 0.07 | 0.45 | 0.79 | 0.72 | 0.61 | 0.3 |
| Yes | Yes | 0.18 | 0.16 | 0.08 | 0.4 | 0.84 | 0.82 | 0.66 | 0.45 |

Table 5.3. Test score for DroNet architecture with input of 160x96 for visually augmented model and non augmented model. Test set is raw Himax frames.

| Pitch Aug. | Visual Aug. | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| No | Yes | 0.56 | 0.34 | 0.16 | 0.58 | -0.48 | 0.44 | -0.28 | 0.03 |
| Yes | Yes | 0.2 | 0.18 | 0.08 | 0.4 | 0.8 | 0.77 | 0.62 | 0.44 |

Table 5.4. Test score for DroNet architecture with input of 160x96 for visually augmented model and non augmented model. Test set is pitch augmented to simulate various pitch angles.

To verify that the cropping was a good simulation for actual pitch change, we collected a dataset with varying pitch angles, while the cart and the subject remained static. We expected to see the model outputs the same prediction for the different pitch values, indicating we reduced the pitch sensitivity. The model behaved as expected in the pitch range it learned during training, as can be seen in Figure 5.10. It is important to mention that for positive pitch values we tilt the camera down, and very soon the head is out of the frame. This hinders the yaw prediction, and can be seen in the graph. The real data collected was far more noisy than the simulated data, as the camera was tilted by hand.

We simulated roll by rotating captured frames and then cropping them (Fig. 5.8), and it is apparent that the pitch augmentation is also beneficial for increasing the tolerance to roll changes (Fig. 5.9) .

We also tested the new model on the Crazyflie, and it showed improvement over the previous models.

## 5.5 Resizing Strategies

As our device was limited in both energy and computational resources, we were looking into using smaller inputs, which would reduce the size of the network and the operations required for inference. We wanted to examine the effects of different methods for size reduction:

- Cropping
- Downsampling with bilinear interpolation
- Downsampling with nearest neighbor interpolation
- Foveating

Naturally, it is not a fair comparison as cropped images would have a smaller FOV. On the other hand, it would benefit from greater details in the center of the images, compared to

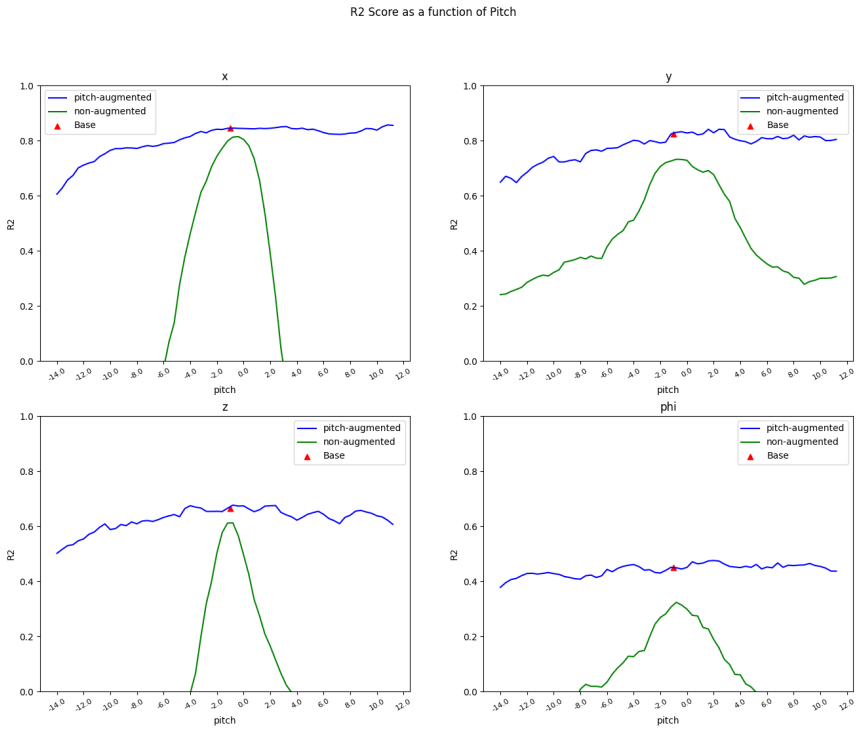


Figure 5.7. $R^2$ score as a function of the pitch angle, for a DroNet architectures with input size 160x96

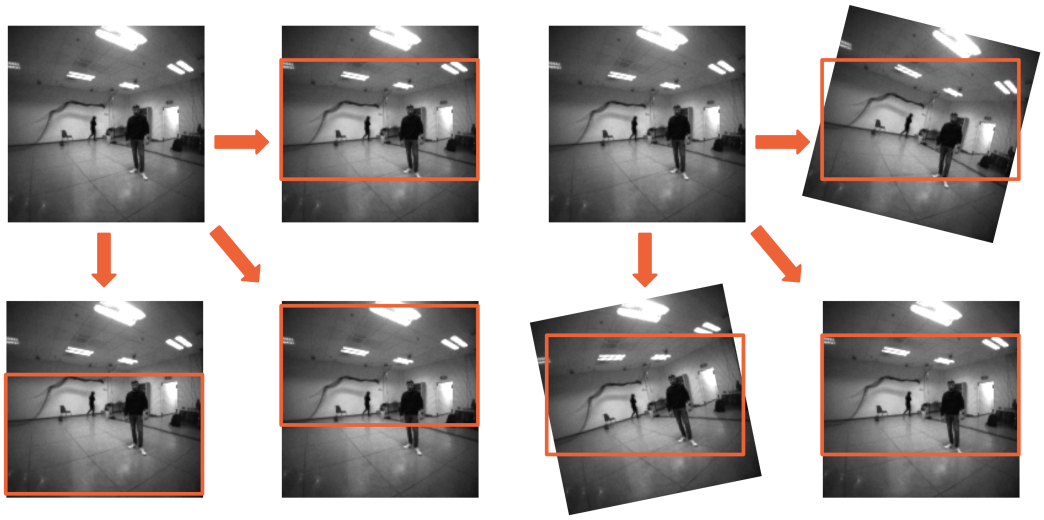

Figure 5.8. $R^2$ Right: Illustration of simulating roll. Left: Illustration of simulating pitch.

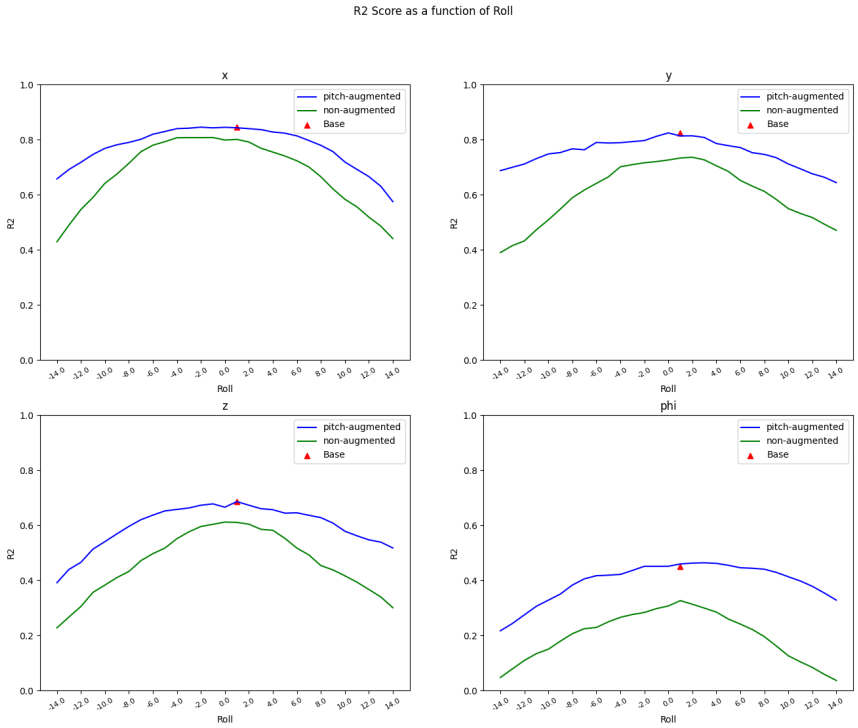


Figure 5.9. $R^2$ score as a function of the roll angle, for a DroNet architectures with input size 160x96

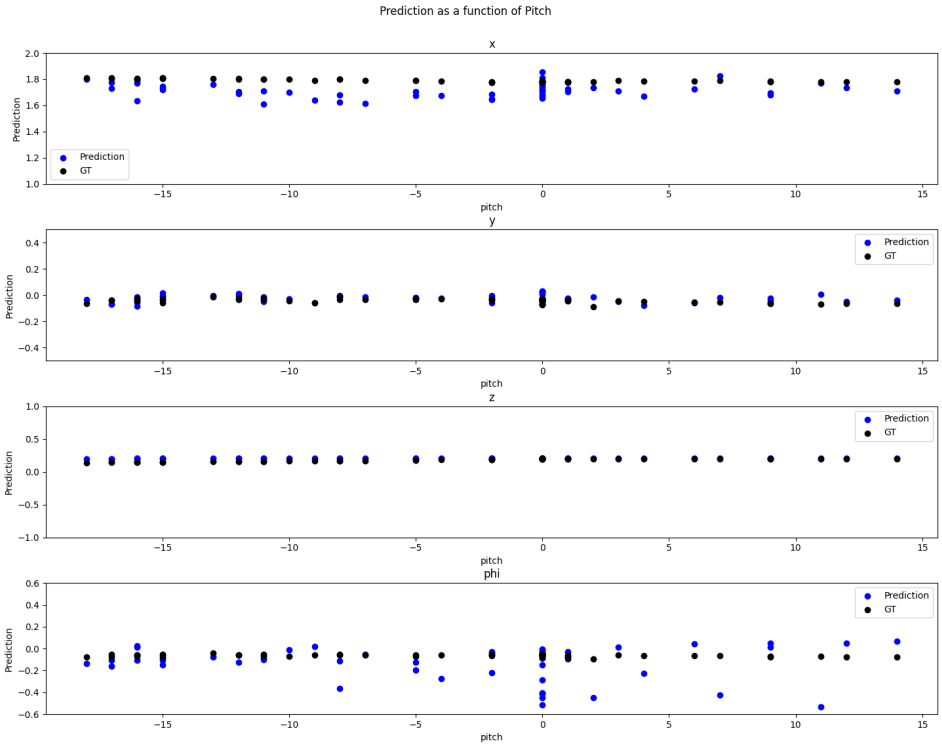


Figure 5.10. Changes in predication as a function of the pitch angle, for a DroNet architectures with input size 160x96

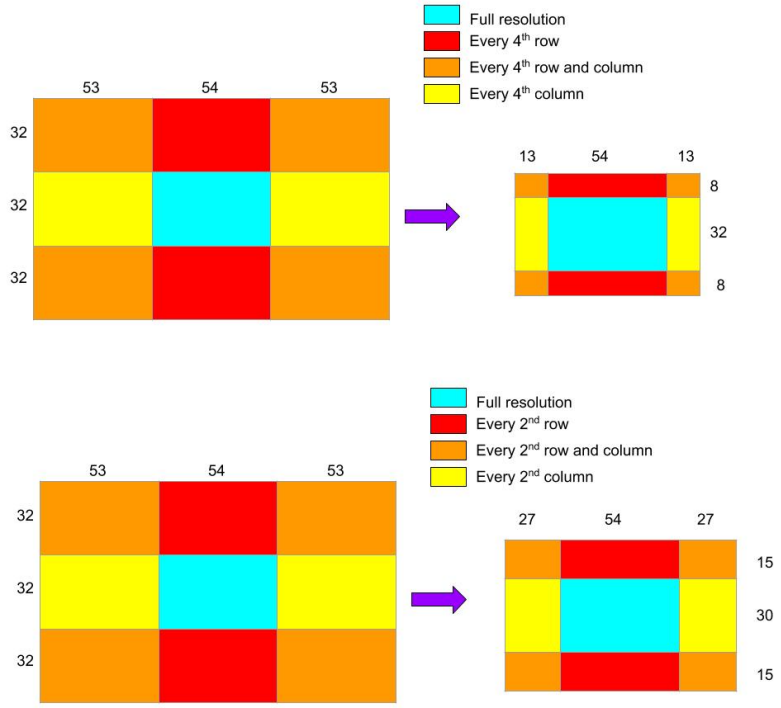


Figure 5.11. Top: Illustration of the foveating process for 160x96 to 80x48. Bottom: Illustration of the foveating process for 160x96 to 80x48.

downsampling. The foveation strategy was designed to profit from both - capturing the full FOV and preserving the full resolution in the center. The foveation scheme is illustrated in Figure 5.11 .

An example of an image down-sampled by our 4 resizing strategies is presented in Figure 5.12. To test the resizing method on performance, we started with the same train and test dataset. For each method, the model was trained and tested on datasets that were resized accordingly. Despite our hope to see the accuracy improve with the foveated scheme, it proved to be less efficient than downsampling, but better than cropping. Downsampling with bilinear interpolation showed no advantage over nearest neighbour interpolation. Considering that bilinear interpolation requires more operations to compute, there is no reason to choose this more sophisticated method, given our limited hardware. We performed the tests for 2 different reduced sizes - 108x60 and 80x48, to verify that results were consistent. The results can be found in Tables 5.5 and 5.6, and Figure 5.13.

## 5.6 Pixel Information

We were interested in finding how much information, or how many pixels, the model requires to make accurate predictions. The first thing that comes to mind is simply resizing the images with the strategies discussed above, and measure the effect of input size on performance. We detected a flaw in that approach - by decreasing the input size, we are also reducing the size of the network. Coupling the two changes would make it hard to conclude if the results are the

Figure 5.12. Example of 4 images produced using the resizing strategies, from the same 160x90 input image (Top-most image). From top left, in clockwise order: Cropping, foveation, nearest neighbor interpolation and bilinear interpolation.

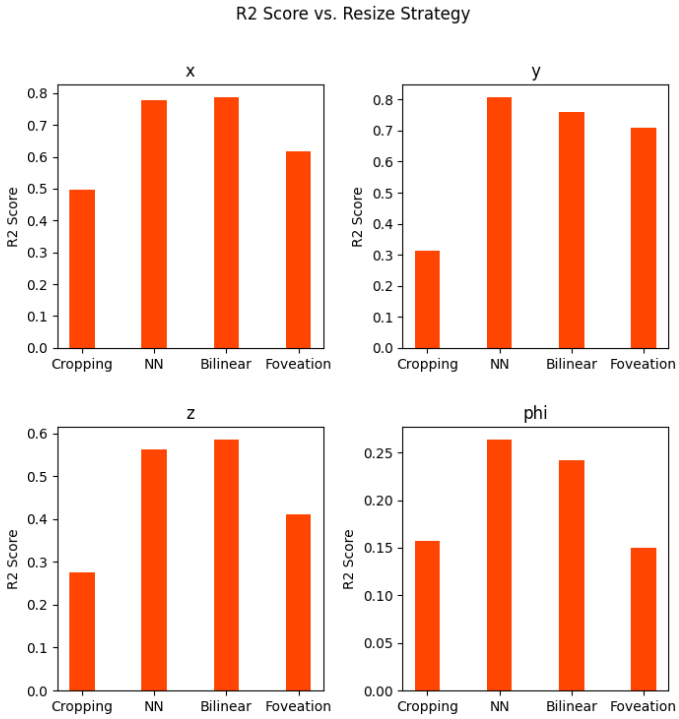


Figure 5.13. Bar plot showing the $R^2$ score for DroNet architecture with input of 80x48 for different resizing strategies

| Resize | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| Cropping | 0.2521 | 0.2433 | 0.0972 | 0.4600 | 0.6979 | 0.5646 | 0.4617 | 0.3159 |
| NN Inter. | 0.2108 | 0.1824 | 0.0875 | 0.4447 | 0.8012 | 0.7933 | 0.5542 | 0.3623 |
| Bilinear Inter. | 0.2158 | 0.1742 | 0.0844 | 0.4299 | 0.7985 | 0.7928 | 0.5966 | 0.3907 |
| Foveation | 0.2495 | 0.1922 | 0.0922 | 0.4593 | 0.7234 | 0.7805 | 0.5233 | 0.3246 |

Table 5.5. Test score for DroNet architecture with input of 108x60 for different resizing strategies

| Resize | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| Cropping | 0.3277 | 0.3101 | 0.1127 | 0.5152 | 0.4977 | 0.3127 | 0.2750 | 0.1569 |
| NN Inter. | 0.2217 | 0.1684 | 0.0927 | 0.4751 | 0.7781 | 0.8083 | 0.5619 | 0.2637 |
| Bilinear Inter. | 0.2172 | 0.1800 | 0.0905 | 0.4863 | 0.7880 | 0.7604 | 0.5859 | 0.2421 |
| Foveation | 0.2970 | 0.2069 | 0.1029 | 0.5220 | 0.6167 | 0.7098 | 0.4117 | 0.1502 |

Table 5.6. Test score for DroNet architecture with input of 80x48 for different resizing strategies

| Pixel Information | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| 160x96 | 0.1850 | 0.1639 | 0.0814 | 0.4006 | 0.8445 | 0.8240 | 0.6652 | 0.4508 |
| 80x48 | 0.1713 | 0.1759 | 0.0795 | 0.3982 | 0.8719 | 0.7638 | 0.6745 | 0.4658 |
| 40x24 | 0.1999 | 0.1831 | 0.0804 | 0.4569 | 0.8140 | 0.7857 | 0.6298 | 0.3184 |
| 20x12 | 0.2658 | 0.2109 | 0.0973 | 0.5168 | 0.6943 | 0.7199 | 0.4666 | 0.1667 |

Table 5.7. Test score for DroNet architecture of size 160x96 with different pixel information

| 48x80 | Parameter Number | MAE | | | | $R^2$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| Reduced Information | 315,364 | 0.17 | 0.17 | 0.08 | 0.39 | 0.87 | 0.76 | 0.67 | 0.46 |
| Reduced Size | 310,756 | 0.22 | 0.17 | 0.08 | 0.45 | 0.77 | 0.78 | 0.6 | 0.3 |

Table 5.8. Test score for DroNet architecture with pixel information of 80x48 vs. input size of 80x48

effect of smaller neural capacity, or lesser amount of pixel information. In order to isolate the impact of pixel information, we decided to keep the input size constant - 160x96. Instead of resizing, we downsample and upsample the image, removing information but preserving the size (Fig. 5.14).

Similarly to previous experiments, we begin with the same dataset of 160x96, which is separated to train and test. For each model, we apply the blurring process on both train and test sets, and eventually compare the results. It appears the pixel information impacts the yaw mostly, which could be explained by lack of facial details. But for other components of the prediction, the low-information models were preforming surprisingly well. The model with 80x48 pixel information was preforming better than then model trained on the original images. The results can be found in Table 5.7.

## 5.7 Input Size

Initially, as we wished to make use of existing datasets, the input size was 108x60. As discussed in the previous section, 80x48 pixels of information seemed enough for accurate prediction, so we trained a model on actual 80x48 sized images, hoping we can decrease the drone's power consumption and/or increase FPS by using a smaller neural network.

However, reducing the model capacity led to a notable performance loss. Even though the number of weights and biases related to the convolution layers remained the same, the number of parameters in the fully connected layer was cut to nearly third, with 1920 parameters for the 160x96 model and 768 for the 80x48 model. The accuracy of the smaller, 80x48 model is still tolerable. Even though the network's size is not dramatically decreased, there are significantly less computations needed to make an inference. Depending on the application, the increased FPS and lower power requirements can be favored over accuracy.

For both input sizes, we noticed that the prediction for the yaw was the least accurate.

Figure 5.14. Top: Original 160x96 input image. Middle: Reduced information image to 80x48. Bottom: Reduced size image to 80x48

| Input Size | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| 80x48 | 0.22 | 0.17 | 0.08 | 0.45 | 0.77 | 0.78 | 0.6 | 0.3 |
| 108x60 | 0.21 | 0.17 | 0.08 | 0.42 | 0.79 | 0.79 | 0.59 | 0.39 |
| 160x96 | 0.18 | 0.16 | 0.08 | 0.4 | 0.84 | 0.82 | 0.66 | 0.45 |

Table 5.9. Test score for DroNet architecture of varying input size

| Architecture | # of Parameters | # of Layers |
|---|---|---|
| PenguiNet | 304,356 | 26 |
| DroNet | 315,364 | 31 |
| ProximityNet | 331,748 | 36 |

Table 5.10. Parameter number for different architectures with input size 160x96

The discussed results are described in Table 5.8. We speculated that for small input sizes we lose crucial information about the user's face, which leads to degraded performance. Other components of the prediction, like the distance from the camera, and the vertical and horizontal offset can be inferred from the user's body, that was still detectable even in low resolutions. To verify our suspicion, we chose a bigger input size, 160x96.

As Table 5.9 suggests, the prediction improves with resolution for all components, but the most drastic impact is on the yaw, which improves by 50% when the resolution is doubled. The linear trend in the yaw is shown in Fig 5.15.

## 5.8 Architecture Exploration

As mentioned previously, throughout the work process we examined 3 architectures. As the project's goal was to port Dario's demo to a nano-drone, the first model we looked into was the ProximityNet. However, since the deployment stage was expected to be complex, the initial goal was to adapt to our task a neural network that was previously deployed. For that, the second model, DroNet, was chosen and modified for our needs. The modified DroNet was successfully deployed using the old pipeline. When we moved to the new Nemo/Dory pipeline, we decided to look into even simpler architectures, in order to reduce the complexity of the conversion process. This led us to PenguiNet.

Since the target platform was a low-power embedded device, we had simplicity and compactness in mind when exploring architectures. Moving from ProximityNet to DroNet, and then PenguiNet, we reduced the number of parameters continuously (Tab. 5.10). However, we still wished to achieve reasonable accuracy, and every architecture was bench-marked to ensure the predictions are precise. DroNet is a well-known architecture that has proven capable of handling complex tasks, and as expected, we suffered mild accuracy loss moving from ProximityNet to DroNet. The transition to PenguiNet was more risky, as the residual connections were removed. We were pleasantly surprised to see that the simplified architecture did provide prediction accuracy at the same level as DroNet. The identity mapping is difficult to learn when stacking non-linearities, and residual connections are known to solve this issue, especially in very deep neural networks. We hypothesize that the effect of the residual connections on very

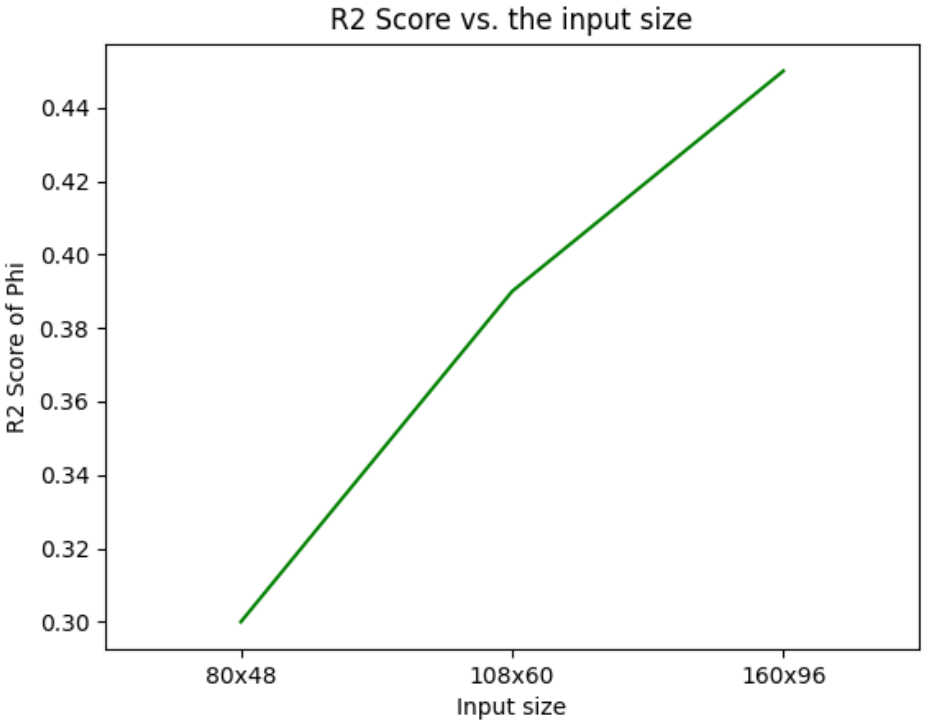


Figure 5.15. The $R^2$ score for the yaw as a function of the input size

| Architecture | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| PenguiNet | 0.2388 | 0.2566 | 0.0939 | 0.4457 | 0.7358 | 0.4778 | 0.5037 | 0.3192 |
| DroNet | 0.2532 | 0.2261 | 0.0923 | 0.4308 | 0.7089 | 0.5999 | 0.5273 | 0.3452 |
| ProximityNet | 0.2224 | 0.2149 | 0.0959 | 0.4004 | 0.7441 | 0.6020 | 0.4676 | 0.4119 |

Table 5.11. Test score for different architectures of size 108x60

| Architecture | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| PenguiNet | 0.1702 | 0.1583 | 0.0711 | 0.3877 | 0.8676 | 0.7973 | 0.7273 | 0.4589 |
| DroNet | 0.1822 | 0.1677 | 0.0795 | 0.3916 | 0.8527 | 0.8139 | 0.6602 | 0.4724 |
| ProximityNet | 0.1483 | 0.1671 | 0.0761 | 0.3272 | 0.8922 | 0.7968 | 0.6937 | 0.5720 |

Table 5.12. Test score for different architectures of size 160x96

shallow networks like ours is negligible, and for this reason the accuracy remains high even when they are removed. The results are given in Tables 5.11 and 5.12.

## 5.9 Hyper-parameters Analysis

After verifying that the PenguiNet architecture is suitable for our task, we proceeded to investigate the impact of hyper-parameters on the accuracy and on-board performance. As presented before, the PULP-Shield and AI Deck have 512KB of L2 memory, and 8MB of RAM (L3) memory. There is a penalty involved in transferring memory, both in time and energy consumption. The transfer between L2 and L3 is the costliest in the system, consuming x10 more energy than L1/L2 transfers.

In the context of edge devices with 3-level memory hierarchy, there are 3 possible categories for neural networks:

I. The entire neural networks, including all weights and required memory allocation for inputs and outputs, can fit in L2.

II. All the components needed for a single layer inference (weights, inputs and outputs) can be stored simultaneously in L2.

III. One layer or more can't fit in L2

Even though PenguiNet is a relatively small neural network, our model of choice, with 160x96 inputs and 32 channels in the first convolution layer, fit the third category. This means, at least one layer cannot be stored entirely in L2, which means we constantly pay a penalty of re-loading weights to L2, and we also need to tile the transfers for the layers that cannot fit in their entirety in L2. But fitting the entire model in L2, we can save on the L2-L3 traffic and even power off the external DRAM, which lowers the power envelope. This can allow us to fly longer time than what we could fly with the L3 powered on, or fly equally long but with improved performance as the spare energy can be leveraged to increase the processor frequency. We examined adjustments to the architecture that would allow PenguiNet to classify for the first

```
osboxes@osboxes:~/Documents/Drone/pulp-frontnet/src/BUILD/GAP8_V2/GCC_RISCV$ size PULPFrontnet
   text    data     bss     dec     hex filename
  60112    4268    3080   67460   10784 PULPFrontnet
```

Figure 5.16. The size of the application binary

category. While it was expected that we lose accuracy in the process of reducing the network's complexity, we wish to see if the performance gain might compensate.

### 5.9.1 Memory Footprint

To estimate which neural networks can fit in L2, we had to make a few assumptions. First, layers are computed sequentially, so we always need to allocated memory for the input and output of a single layer. This is because PenguiNet does not include residual connections. Second, the camera capture can be done asynchronously, which means after it was given as an input to the first layer, the buffer is used to capture the next frame. So this memory cannot be used for intermediate allocations during inference. Since the captured image size is 160x160 (before cropping), we decided to put aside 25KB for the camera buffer. Third, memory is used for other functionalities, such communication with the firmware and the code itself, which is around 70KB (Fig. 5.16).

We decided to limit ourselves to 400KB of L2 for all the inference memory requirements. To compute the memory footprint of a PenguiNet model, we used the following formulas for computing the filter, input and output buffer sizes for a given layer:

$$\text{Input} = w \times h \times c_{in} \tag{5.2}$$

$$\text{Output} = w \times h \times c_{out} \tag{5.3}$$

$$\text{Filter} = k \times c_{in} \times c_{out} \tag{5.4}$$

Where

- k - size of the convolution kernel
- h - input height
- w - input weight
- $c_{in}$ - number of input channels
- $c_{out}$ - number of output channels

In order to compute whether a model can classify as category 2, we need to verify that:

$$\max_{l \in \text{layers}} (\text{input}_l + \text{output}_l + \text{filter}_l) < 400KB \tag{5.5}$$

However, if we want to fit to entire model in L2, which classifies as category 1, we need to verify that:

$$\sum_{l \in \text{layers}} \text{filter}_l + \max_{l \in \text{layers}} (\text{input}_l + \text{output}_l) < 400KB \tag{5.6}$$

In the Nemo-Dory pipeline the inputs, weights and biases are 8-bit. The batch-norm and activations are 32-bits, and in a precise memory analysis should be computed as 4 bytes per elements.

| $h$ | $w$ | $c_{in}$ | Weights | Max. layer memory | Total memory |
|---|---|---|---|---|---|
| 48 | 80 | 32 | 298784 | 122880 | 421664 |
| 96 | 160 | 16 | 77968 | 245760 | 323728 |
| 96 | 160 | 32 | 303392 | 491520 | 794912 |
| 96 | 160 | 64 | 1196608 | 983040 | 2179648 |

Table 5.13. Memory footprint of the chosen configurations.

But as we take a considerable buffer by reserving 112Kb for all non-inference functionalities, we can simplify our computations by treating all elements as 1 byte.

The four configurations we decided to explore are presented in Table 5.13. A more detailed review of the memory footprint of each configuration can be found in the Tables 5.14, 5.15, 5.16 and 5.17.

| name | $k$ | $h$ | $w$ | $c_{in}$ | $c_{output}$ | *in space [B]* | *out space [B]* | *filter space [B]* | *computation [MAC]* | *CCR [MAC/B]* | *layer memory* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *conv1* | 5 | 96 | 160 | 1 | 64 | | 983040 | 1600 | 24576000 | 24.95 | 983040 |
| *conv2* | 3 | 24 | 40 | 64 | 64 | 61440 | 61440 | 36864 | 35389440 | 221.53 | 122880 |
| *conv3* | 3 | 12 | 20 | 64 | 64 | 15360 | 15360 | 36864 | 8847360 | 130.9 | 30720 |
| *conv4* | 3 | 12 | 20 | 64 | 128 | 15360 | 30720 | 73728 | 17694720 | 147.69 | 46080 |
| *conv5* | 3 | 6 | 10 | 128 | 128 | 7680 | 7680 | 147456 | 8847360 | 54.33 | 15360 |
| *conv6* | 3 | 6 | 10 | 128 | 256 | 7680 | 15360 | 294912 | 17694720 | 55.65 | 23040 |
| *conv7* | 3 | 3 | 5 | 256 | 256 | 3840 | 3840 | 589824 | 8847360 | 14.8 | 7680 |
| *fc1* | 1 | 1 | 1 | 3840 | 4 | 3840 | 4 | 15360 | 15360 | 0.79 | 3844 |

Table 5.14. Memory footprint analysis for a PenguiNet with input size 160x96 and 64 channels.

| name | $k$ | $h$ | $w$ | $c_{in}$ | $c_{output}$ | *in space [B]* | *out space [B]* | *filter space [B]* | *computation [MAC]* | *CCR [MAC/B]* | *layer memory* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *conv1* | 5 | 96 | 160 | 1 | 32 | | 491520 | 800 | 12288000 | 24.95 | 491520 |
| *conv2* | 3 | 24 | 40 | 32 | 32 | 30720 | 30720 | 9216 | 8847360 | 125.21 | 61440 |
| *conv3* | 3 | 12 | 20 | 32 | 32 | 7680 | 7680 | 9216 | 2211840 | 90 | 15360 |
| *conv4* | 3 | 12 | 20 | 32 | 64 | 7680 | 15360 | 18432 | 4423680 | 106.66 | 23040 |
| *conv5* | 3 | 6 | 10 | 64 | 64 | 3840 | 3840 | 36864 | 2211840 | 49.65 | 7680 |
| *conv6* | 3 | 6 | 10 | 64 | 128 | 3840 | 7680 | 73728 | 4423680 | 51.89 | 11520 |
| *conv7* | 3 | 3 | 5 | 128 | 128 | 1920 | 1920 | 147456 | 2211840 | 14.61 | 3840 |
| *fc1* | 1 | 1 | 1 | 1920 | 4 | 1920 | 4 | 7680 | 7680 | 0.79 | 1924 |

Table 5.15. Memory footprint analysis for a PenguiNet with input size 160x96 and 32 channels.

| name | $k$ | $h$ | $w$ | $c_{in}$ | $c_{output}$ | *in space [B]* | *out space [B]* | *filter space [B]* | *computation [MAC]* | *CCR [MAC/B]* | *layer memory* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *conv1* | 5 | 96 | 160 | 1 | 16 | | 245760 | 400 | 6144000 | 24.95 | 245760 |
| *conv2* | 3 | 24 | 40 | 16 | 16 | 15360 | 15360 | 2304 | 2211840 | 66.97 | 30720 |
| *conv3* | 3 | 12 | 20 | 16 | 16 | 3840 | 3840 | 2304 | 552960 | 55.38 | 7680 |
| *conv4* | 3 | 12 | 20 | 16 | 32 | 3840 | 7680 | 4608 | 1105920 | 68.57 | 11520 |
| *conv5* | 3 | 6 | 10 | 32 | 32 | 1920 | 1920 | 9216 | 552960 | 42.35 | 3840 |
| *conv6* | 3 | 6 | 10 | 32 | 64 | 1920 | 3840 | 18432 | 1105920 | 45.71 | 5760 |
| *conv7* | 3 | 3 | 5 | 64 | 64 | 960 | 960 | 36864 | 552960 | 14.25 | 1920 |
| *fc1* | 1 | 1 | 1 | 960 | 4 | 960 | 4 | 3840 | 3840 | 0.79 | 964 |

Table 5.16. Memory footprint analysis for a PenguiNet with input size 160x96 and 16 channels.

| name | $k$ | $h$ | $w$ | $c_{in}$ | $c_{output}$ | *in space [B]* | *out space [B]* | *filter space [B]* | *computation [MAC]* | *CCR [MAC/B]* | *layer memory* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *conv1* | 5 | 48 | 80 | 1 | 32 | | 122880 | 800 | 3072000 | 24.83 | 122880 |
| *conv2* | 3 | 12 | 20 | 32 | 32 | 7680 | 7680 | 9216 | 2211840 | 90 | 15360 |
| *conv3* | 3 | 6 | 10 | 32 | 32 | 1920 | 1920 | 9216 | 552960 | 42.35 | 3840 |
| *conv4* | 3 | 6 | 10 | 32 | 64 | 1920 | 3840 | 18432 | 1105920 | 45.71 | 5760 |
| *conv5* | 3 | 3 | 5 | 64 | 64 | 960 | 960 | 36864 | 552960 | 14.25 | 1920 |
| *conv6* | 3 | 3 | 5 | 64 | 128 | 960 | 1920 | 73728 | 1105920 | 14.43 | 2880 |
| *conv7* | 3 | 2 | 3 | 128 | 128 | 768 | 768 | 147456 | 884736 | 5.93 | 1536 |
| *fc1* | 1 | 1 | 1 | 768 | 4 | 768 | 4 | 3072 | 3072 | 0.79 | 772 |

Table 5.17. Memory footprint analysis for a PenguiNet with input size 80x48 and 32 channels.

| Hyper-parameters | | | MAE | | | | $R^2$ Score | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $h$ | $w$ | $c_{in}$ | $x$ | $y$ | $z$ | $\phi$ | $x$ | $y$ | $z$ | $\phi$ |
| 48 | 80 | 32 | 0.2195 | 0.1746 | 0.0878 | 0.4791 | 0.7857 | 0.7791 | 0.5865 | 0.2642 |
| 96 | 160 | 16 | 0.1922 | 0.1674 | 0.0869 | 0.4285 | 0.8417 | 0.7607 | 0.6070 | 0.3704 |
| 96 | 160 | 32 | 0.1727 | 0.1552 | 0.0721 | 0.4037 | 0.8625 | 0.7875 | 0.7174 | 0.4470 |
| 96 | 160 | 64 | 0.1689 | 0.1597 | 0.0735 | 0.3651 | 0.8713 | 0.7850 | 0.7103 | 0.5050 |

Table 5.18. Prediction accuracy for PenguiNet with varying hyper parameters

### 5.9.2 Analysis

The models with varying hyper-parameters were trained and bench-marked on the same dataset (up to resizing) and with the same infrastructure. As expected, networks with higher complexity achieved better results on the test set, and the most affected component is the yaw (Tab. 5.18). In order to measure the performance, the full precision PyTorch models were deployed using the NEMO-DORY pipeline. The largest model, 160x96x64 (w,h,c), failed to deploy as it violated the memory constraint on L1. The other three models were deployed successfully. In the current version of Dory, optimization for models from category (I) is not yet implemented. Even if the model can fit entirely in L2, the weights will be copied from L3 every iteration. Using the DORY-generated code, we implemented the L2 optimization by ourselves. The evaluation is not covered in this work (but will be part of a paper), and we focus on analysing the estimate contribution of such optimization.

L3-L2 communication is costly in terms of energy, and we wanted to evaluate how much can we save by avoiding the repetitive transfers in favor of copying all the weights once, in the very beginning. Not only we would save power on the transfer, we could also switch off the L3. Since there is not direct way to measure the power consumption for memory access, we decided to count how many cycles are required for copying the weights every iteration. The cycles can be converted to time like so:

$$t = \frac{\text{cycles}}{\text{CLK}_{\text{freq}}} \tag{5.7}$$

We can compute the total energy cost per iteration as

$$E_{\text{total}} = t \times P_{\text{transfer}} + (T - t) \times P_{\text{standby}} \tag{5.8}$$

Where T is the overall iteration time, including image capture, inference and communication. It was measured previously that T=70ms. The DMA is affected by the clock frequency of the relevant processor. L3-L2 transfers are coordinated by the $\mu$-DMA, so the relevant frequency would be that of the fabric controller, which was set to 100MHz. Using the memory's data-sheet [CYPRESS], we can infer how much power is required for transfers and standby.

$$P_{\text{transfer}} = V_{\text{cc}} \times I_{\text{hyperRAM read}} = 1.95V \times 20.1mA = 39.195mW \tag{5.9}$$

$$P_{\text{standby}} = V_{\text{cc}} \times I_{\text{standby}} = 1.95V \times 160\mu A = 312\mu W \tag{5.10}$$

These computations are for the typical current, but the worst case scenario may require up to 3 times larger current putting $I_{\text{hyperRAM read}}$ at 60.3mA. The leakage current was neglected in the computation for simplicity and because it is not very significant (max. 2mA). Also, efficiency of

| Hyper-parameters | | | Theoretical time (ms) | Measured time (ms) | Energy consumption ($\mu J$) |
|---|---|---|---|---|---|
| $h$ | $w$ | $c_i n$ | | | |
| 48 | 80 | 32 | 3.06 | 4.32 | 189.8 |
| 96 | 160 | 16 | 0.79 | 1.23 | 69.6 |
| 96 | 160 | 32 | 3.1 | 4.39 | 192.5 |

Table 5.19. energy consumption analysis for a PenguiNet models with varying hyper-parameters.

the voltage regulator is assumed to be 100% although it is 86% for the same reasons. Even in the average case, the power cost required for reading from the HyperRam ( 39.2mW) is large - the average power consumption by the application, on both fabric controller and cluster is 32mW. The peak consumption for HyperRAM read is $\sim$ 117mA, compared to peak consumption of 44mW in the application.

To verify the measurements are reasonable, we first computed the theoretical (and ideal) time it would take to copy all the weights from L3 to L2. The hyper bus interface is 8-bit wide, and its bandwidth depends on the fabric controller frequency:

$$\text{BW} = \text{width} * \text{CLK}_{\text{freq}} = 1B \times 100MHz = 100MB/s \tag{5.11}$$

So the time to transfer memory would be:

$$t = \frac{\text{memory in bytes}}{\text{BW}} \tag{5.12}$$

The results of these measurements are brought in Table 5.19.

Energy cost for memory transfers is up to 13% of the application energy consumption, which is $\sim$ 1500$\mu$J per iteration. It is important to note that the application implementation is not optimized - for example, the frame capture and inference are sequential. The relative gain in energy consumption will be more prominent for an optimized application with shorter iteration time.

The most important advantage of avoiding the data transfers is the reduction of the power envelope. Instead of of average overall power consumption of 71.2mW for both computation and memory in the inference stage, we can only look at max($P_{\text{mem}}, P_{\text{app}}$)= 39.2mW.

In summary, memory transfers between L3 and L2 are costly. Since the GAP8 is not very fast, our application in compute-bound and not memory bound. In cases where the application is bound by the bandwidth, it is advisable to use smaller neural networks that fit entirely in L2 to tackle the bottleneck. Also, when the overall power supply is low, performing the L3-L2 transfer before the inference loop can reduce the power envelop.

# Chapter 6

# Conclusion and Future Work

In this work, we demonstrated that it is possible to port a PyTorch convolutional neural network to a resource-constrained embedded device, while maintaining the quality of results. As discussed in the final evaluation, the performance on the selected task was satisfying, but there is room for improvement and additional research in several different directions.

**User-Oriented Improvements** - A pipeline that allows deploying PyTorch deep neural networks to ultra low power platforms provides chance to explore in multiple directions. While the current pipeline can be used by an experienced Python developer, it still involves quite a bit of tweaking and manual adjustments, and the most immediate improvement is simplifying this process and providing a better user interface, possibly a graphic one. Restructuring the code as a self-contained, intuitive tool can increase its appeal among the scientific community and help drive forward research.

**Data-set Expansion** - The model was trained and tested on fairly limited data-sets. It would be beneficial to collect more varied data, such as recording different people. Another possibility is to try and capture images outside the Drone Arena, and enable the application to operate in other indoor environments. Recording in other rooms, we will be missing the tracking information provided by the OptiTrack system, but this challenge may be overcome using semi-supervised learning techniques.

**System Inputs** - The current implementation only receives the video frames as inputs, and a possible research direction might be considering additional information during inference, such as the drone's attitude and odometry.

**State-Awareness** - The NEMO/DORY pipeline supports only convolutional neural networks at the present, which infer per frame. Unlike recurrent neural network, convolutional neural networks do not consider the state of the system. Implementing support for networks that can process sequences, or adding a state-aware component on top of the convolutional neural network, can also achieve better accuracy.

**Online Learning** - Another interesting direction is learning on the fly. Easily adapting to new environments, by introducing new data in an incremental way, can aid in popularizing the usage of nano-drones for autonomous tasks.

# Bibliography


bitcraze. Crazyflie Hardware Specifications. URL https://store.bitcraze.io/products/crazyflie-2-1.

Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus, 2020.

Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks, 2018.

Francesco Conti. Technical report: Nemo dnn quantization for deployment model, 2020.

CYPRESS. HyperFlash and HyperRAM Multi-Chip Package Datasheet. URL https://www.cypress.com/file/322936/download.

ETH/UNIBO. PULP Home Page. URL http://iis-projects.ee.ethz.ch/index.php/PULP.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Himax. Himax Hardware Specifications. URL https://www.himax.com.tw/products/cmos-image-sensor/image-sensors/hm01b0/.

Wolfgang Hönig and Nora Ayanian. *Flying Multiple UAVs Using ROS*, pages 83–118. Springer International Publishing, 2017. ISBN 978-3-319-54927-9. doi:10.1007/978-3-319-54927-9_3. URL https://doi.org/10.1007/978-3-319-54927-9_3.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

A. Loquercio, A. I. Maqueda, C. R. del-Blanco, and D. Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 2018.

Dario Mantegazza, Jérôme Guzzi, Luca Maria Gambardella, and Alessandro Giusti. Vision-based control of a quadrotor in user proximity: Mediated vs end-to-end learning approaches. *2019 IEEE International Conference on Robotics and Automation (ICRA)*, 2019. doi:10.1109/ICRA.2019.8794377. URL https://github.com/idsia-robotics/proximity-quadrotor-learning.

Hanna Mueller. Design of a PULP-Shield for Autonomous nano-UAVs. Januray 2018. URL http://iis-projects.ee.ethz.ch/index.php/PULP-Shield_for_Autonomous_UAV.

OptiTrack. 12 Cameras Setup. URL https://optitrack.com/systems/#robotics/primex-13/12.

D. Palossi, F. Conti, and L. Benini. An open source and open hardware deep learning-powered visual navigation engine for autonomous nano-uavs. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 604–611, May 2019a. doi:10.1109/DCOSS.2019.00111.

D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini. A 64mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 2019b. ISSN 2327-4662. doi:10.1109/JIOT.2019.2917066.

Parrot. Bebop 2. URL https://www.parrot.com/us/drones/parrot-bebop-2.

PULP. PULP SDK. URL https://github.com/pulp-platform/pulp-sdk.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

GreenWaves Technologie. AI Deck. URL https://greenwaves-technologies.com/ai-deck-is-available-in-early-access/.