Learning-Based Relative State Estimation for Ultralow-Power Multirotor Teams

Thesis in partial fulfillment of the requirements for the degree
Master of Science (M.Sc.)

in the course of studies
ICT Innovation (Embedded Systems)

submitted by
Keerthana P Laxmish
Matriculation number : 478811

October 26, 2023

Technische Universität Berlin
Faculty IV - Institute of Computer Engineering and Microelectronics
Institute of Computer Engineering and Microelectronics
Intelligent Multi-Robot Coordination Lab

Declaration

Statement of authorship

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgefuhrten Quellen und Hilfsmittel angefertigt habe.

[illegible]

Berlin, October 26, 2023

Acknowledgements

I would like to express my gratitude to Akmaral Moldagalieva and Pia Hanfeld for their excellent mentoring and unwavering support through the thesis journey. The patience and encouragement shown by them was instrumental in shaping the thesis. I would also like to thank the entire team at the IMRC Lab, TU Berlin for providing a positive environment, for both learning and fun. My journey at the IMRC Lab has been one of not only academic development but also significant personal growth, thanks to the welcoming atmosphere at the lab and the diverse conversations during breaks. A special thank you to Ilaria Cicchetti-Nilsson for being a constant source of motivation and incredible moral support throughout my thesis. Final words of gratitude go to Shri, for being my pillar of strength, through thick and thin.

Learning-Based Relative State Estimation for Ultralow-Power Multirotor Teams

Abstract : Multirotor teams have found applications in various fields due to their efficiency, versatility, and autonomous behavior. A critical aspect of achieving autonomy in such teams is the ability to perform relative localization between team members without relying on external positioning systems. Deep neural networks that learn from monocular camera images have shown success in achieving this. However, the challenge lies in deploying these memory and computation-intensive models onboard ultra low-power multirotors. In this work, a vision-based deep neural network that predicts the relative positions of neighboring robots using grayscale images is quantized and optimized. Open source dataset of close-proximity images and software tools are used to automate the end-to-end deployment on a resource constrained multi-core system-on-chip. Four variants of the model with differing input image resolutions are deployed and their performance in terms of prediction quality and inference rate is analyzed. The evaluations indicate that all four minimized variants maintained their performance close to that of the unconstrained model. The models with lowest input resolutions exhibit approximately a 36% higher error in relative localization compared to the models with higher input resolutions. However, using lower input resolutions enables a roughly $60-70\%$ increase in the inference rate. To determine the optimal choice of input image resolution, it is crucial to strike the right balance between the two factors based on the application.

Zusammenfassung : Multirotor-Teams finden aufgrund ihrer Effizienz, Vielseitigkeit und ihres autonomen Verhaltens in verschiedenen Bereichen Anwendung. Ein entscheidender Aspekt für die Autonomie solcher Teams ist die Fähigkeit, die Position benachbarter Multirotoren im Team zu lokalisieren, ohne sich auf externe Positionierungssysteme verlassen zu müssen. Tiefe neuronale Netze konnten in der Vergangenheit bereits erfolgreich eingesetzt werden, um die relative Position von benachbarten Multirotoren mit Hilfe von Kamerabildern zu bestimmten. Die Herausforderung besteht jedoch darin, diese speicher- und rechenintensiven Modelle an Board eines Multirotors mit sehr geringem Stromverbrauch einzusetzen. Diese Arbeit befasst sich mit der Quantisierung und der Inbetriebnahme eines neuronalen Netzes, dass die Position von benachbarten Robotern anhand von schwarz-weiß Bildern vorhersagt. Das Ziel ist es, die Berechnung an Board in Echtzeit durchzuführen. Open-Source-Datensätze von Nahaufnahmen und Software-Tools werden verwendet, um die End-to-End-Bereitstellung auf einem ressourcenbeschränkten Multi-Core-System-on-Chip zu automatisieren. Es wurden vier Varianten des Modells mit unterschiedlichen Auflösungen des Eingangsbildes eingesetzt, und ihre Leistung in Bezug auf die Vorhersagequalität und die Inferenzrate wird analysiert. Die Auswertungen zeigen, dass alle vier minimierten Varianten ihre Leistung nahe an der des ungebundenen Modells halten. Die Modelle mit der niedrigsten Eingabeauflösung weisen im Vergleich zu den Modellen mit der höchsten Eingabeauflösung einen um etwa 36% höheren relativen Lokalisierungsfehler auf. Die Verwendung niedrigerer Eingangsauflösungen ermöglicht jedoch eine Steigerung der Inferenzrate um etwa $60 - 70\%$. Um die optimale Wahl der Eingangsbildauflösung zu bestimmen, ist es entscheidend, je nach Anwendung das richtige Gleichgewicht zwischen den beiden Faktoren zu finden.

First Thesis Examiner : Asst. Prof. Dr. Wolfgang Hönig
Title: Head of Intelligent Multi-Robot Coordination Lab

Second Thesis Examiner : Prof. Dr. Oliver Brock
Title: Head of Robotics and Biology Laboratory

Thesis Supervisors :
Akmaral Moldagalieva, PhD student
Pia Hanfeld, PhD student

Contents

List of Figures

List of Tables

1 Introduction

Unmanned Aerial Vehicles (UAVs) are aerial vehicles that do not carry a human operator. UAVs can be controlled either autonomously by an onboard computing system or by a remote ground-based system. Due to their size and agility, they can be used in various applications where human intervention is either impossible or hazardous. UAVs have found applications in fields such as agriculture (for crop monitoring), search and rescue (providing assistance in hazardous scenarios), mapping (for land surveying and geographic information systems (GIS)), and delivery and logistics, among others.

With the revolutions in the electronic device industry enabling the integration of sensors, actuators, and microcontrollers on miniaturized battery-powered devices, micro UAVs (referred to as MAVs) have garnered interest from both public and private research laboratories. MAVs are specialized for tasks that require close-range operation or navigation in narrow spaces that are difficult for larger UAVs to access. A swarm of such MAVs can perform tasks that would be challenging for an individual or a centralized control system. To interact with other MAVs in the vicinity and perform cooperative tasks autonomously without collisions or interference, the MAVs must be capable of localization, i.e., they should have information about their own state as well as the states of their neighbors.

Localization techniques using Global Navigation Satellite System (GNSS) [18] or Ultra-Wide Band (UWB) communication [13] have been proposed. However these methods may suffer from latency and high power consumption due to the remote communication, require an external system and are not scalable to swarms. For swarms in GPS-denied environments, localization methods have to be real-time and onboard the MAVs. Moreover these methods must adhere to the strict power and computation constraints of the devices. In recent times deep neural networks have been proved to show good performance for vision-based tasks. Therefore, this thesis focuses on implementing a vision-based learning method for relative localization on an open-source robotic platform and conducting a study on the real-time onboard performance.

The scientific contributions in the thesis are listed below:

1. Quantization of a deep neural network that performs multi-robot relative localization.

2. Deploying the quantized model on a robotic platform for real-time inference.

3. Experimental evaluation and comparative analysis of the prediction quality and achievable inference speed based on different input resolutions.

The structure of the thesis is as follows. Chapter 2 introduces relative localization for autonomous behavior of multirotor teams. A literature review on the different methods used, the hardware architectures of these devices and end-to-end deployment of learning based state estimation methods are detailed. A short background of the chosen deployment flow and open source platforms is also discussed. Chapter 3 describes the details about the implementation of a vision-based deep neural network model and its quantization to an integer representation. The deployment of the quantized model onto the chosen robotic platform is discussed. Chapter 4 presents the experiments performed and the results of a comparative analysis. Chapter 5 summarizes the work done in the thesis and outlines future work in the field.

2 Background

The following section introduces the relative localization methods for autonomous multirotor teams and the development flow for deploying one such method on-board an ultralow power processor.

2.1 Relative localization of UAVs

Multiple methods of localization for UAVs have been explored in research. Some of them include using motion capture system [30] for indoor aerial UAVs or Global Positioning System (GPS) providing high accuracy with low computational complexity. Such external systems may not always be available and they are also susceptible to jamming and interference in environment with multiple obstacles. Alternative methods are needed that do not rely on an external system, that are not limited to indoor operations and can operate in GPS-denied environments. Relaxing the localization requirement from absolute to relative also provides more flexibility. Relative localization is the estimation of the MAV's location in relation to its environment, another MAV or a reference point.

With the objective of eliminating the need for external systems and making relative localization possible for real-time operations, different sensor-based approaches were designed. One such approach was to develop a novel on-board infrared 3D relative positioning sensor that could provide proximity sensing in 3D space. This sensor was proven to enable inter-robot spatial co-ordination in indoor environment without using complex algorithms and relying on indoor illumination [22]. Audio-based localization wherein one chirping MAV (positioning beacon) flies around the other MAVs enabling the observers to measure their relative position to the beacon using the recorded audio from their on-board microphone array

[1] has also emerged in research. Other methods include relative localization by using wireless communication between the UAVs in a swarm, to exchange position and orientation information and fusing the received data with the range measurements from onboard antennas [13].

The sensor-based approaches mentioned above, while successful, either require large sensors to be mounted or, even if they utilize smaller communication chips, suffer from bandwidth limitations and poor performance as the swarms grow larger or if the environment is cluttered.

An efficient relative localization technique that is scalable for autonomous swarms of ultralow-powered MAVs is required. Vision sensors are light-weight, versatile and offer rich information from their data. Moreover they can be easily combined with the deep learning for different applications. These sensors can be mounted on the MAVs and hence offer real-time capabilities without need for bandwidth limited communication mechanisms. Thus vision-based approaches to generate stable and safe flight paths seem promising both in terms of performance and scalability [7].

Vision-based relative localization

Vision-based approaches for autonomous UAVs vary based on the size of the UAVs and their onboard embedded computing ability. UAVs that are larger than 0.5 kg can afford to work with high resolution images and run computationally intensive algorithms for perception, feature extraction, localization and mapping. For the pocket size UAVs that generally weigh less than 100 g, the computing ability is limited to ultralow-power microcontrollers and usage of low resolution images. Here, a single, light-weight and energy-efficient camera is used to gather information about the surroundings. Vision-based methods are also scalable for multi-robot localization, unlike the previously mentioned approaches that depend on external positioning systems.

Classical approaches like vision-based SLAM (Simultaneous Localization and Mapping) algorithms have been developed to obtain the pose estimates for the MAVs using on-board sensor data, which are then used by a PID controller to perform a flight of computed way points [14] [24]. Visual SLAM algorithms require significant computational resources for real-time operation, thus straining the onboard computing hardware, potentially leading to delays in processing and control. These methods sometimes employ off-board computations relying on wireless communication such as WiFi which have constrained communication channel bandwidth and performance drop in cluttered surroundings.

Visual marker-based relative localization have also been successfully implemented for collision avoidance. Here, the system consists of a Hummingbird quadcopter with a small ARM-based onboard computer, and a large red marker with two built in fish-eye cameras. There is an additional white marker on the red one for tracking by the 3D motion capture system. Standard computer vision algorithm and color segmentation to identify the red marker accomplishes the quadcopter detection task, i.e., the size, position and shape of the red marker from the fish-eye camera image can be used to compute the 3D position of the UAV relative to the camera. Once the quadcopters are localized, the information was used to trigger collision avoidance [23]. These methods however require special hardware extensions to the already small UAVs which could degrade the performance and autonomy of the UAV.

With Deep Neural Networks (DNNs) being successfully implemented for computer vision tasks, marker-less learning-based visual localization could potentially overcome the disadvantages of the previously listed methods. Learning-based methods have been implemented for multi-robot relative localization [12], autonomous aerial interception [29], visual tracking[19], and prediction of dangerous aerodynamic effects [15] among others.

The general principle behind the vision-based approach for relative localization is described in the following section. The translation of the captured 2D camera image of an object into its real-world 3D position requires modeling of the camera behavior. First three different frames are defined, image frame **I** with top left origin, camera frame **C** to represent the 3D position of a UAV with respect to the camera and the robot frame **R** that is fixed to the UAV (with a vertical x-axis, z- and y-axes pointing forward and left). The camera intrinsic matrix **K** defines the translation of a UAV's position from **R** into its own camera frame **C** as below:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where f_x and f_y are the focal lengths and c_x and c_y are principle points of the camera. The coordinates represented in **C** can be transformed into **I** as follows:

$$\mathbf{p}^I = K\mathbf{p}^C \tag{2.2}$$

where $\mathbf{p}^I = [u, v, 1]$ are position vector in image frame with pixel coordinates (u,v) representing the center of the detected neighbor and $\mathbf{p}^C = [x_c, y_c, z_c]$ is the position vector in camera frame.

The 3D relative position of a neighbor with respect to the camera frame can be obtained by inverting eq. (2.2) [15].

$$\mathbf{p}^C = \left[z_c \frac{(u - c_x)}{f_x}, z_c \frac{(v - c_y)}{f_y}, z_c \right] \tag{2.3}$$

The camera intrinsic parameters can be estimated by a calibration process with a checkerboard pattern. Given an image of the neighboring robots and the distance to the neighbors from the camera, the 3D position of the neighbors in the camera frame of the observing UAV can be obtained [12].

2.2 Machine Learning and Neural Networks

The past decade has seen the emergence of Artificial Intelligence (AI), a field that focuses on teaching computers to perform tasks by imitating human behavior. Machine Learning (ML) and its subset Deep Learning (DL) are some of the methods designed to achieve this objective. Essentially these methods perform pattern recognition to learn key features of the task at hand, similar to the way human beings learn problem solving. Given a set of data, Machine Learning as a field works on developing algorithms that approximate a function to represent the data using different types of learning methods. One such learning method is

Supervised Learning where the objective is to infer patterns from a given dataset and enable the ML model to learn to map the features to their respective targets so that these algorithms can be reused for making the feature-target predictions on new unknown data (inference). The dataset is split into training and testing dataset. The performance of the algorithm can be tuned at every iteration on a smaller dataset (validation dataset) that is a subset of the training dataset. The performance of the model can be evaluated by testing the prediction ability on a new unknown dataset (testing dataset). While the validation dataset contains images in the same domain as the training dataset, the testing dataset contains additional instances of the original dataset that have not been seen by the training algorithm, to ensure that the model learns a generalized behavior and is not specific to the training set. Supervised learning methods are commonly used for tasks such as regression and classification. Regression is the task of predicting numeric data whereas classification is the task of predicting the category of data. An example of the regression task could be to predict the price of a certain item given the data about trends in the past prices. A popular example for the classification task is the prediction of the handwritten digits from a given image.

Artificial Neural Network (ANN) is a machine learning algorithm that closely tries to mimic the behavior of the human brain. In the human brain, information is processed by the neurons which are connected to each other via a synapse. In a similar way, ANN contains nodes that pass information to other nodes via connections which are weighted towards a desirable outcome. ANNs have a layer of input nodes and a layer of output nodes, connected by some hidden layers in between[4].

The fundamental building block of ANNs is a perceptron. It mimics the biological neuron by taking in multiple input values, performing a weighted sum of the inputs and then passing it to a non-linear function (activation) so that a particular feature can be extracted from the input. A perceptron with three inputs as shown in fig. 2.1 can be represented by the following function:

$$y_j = f(\sum_{i=1}^{3} W_{ij} \times x_i + b) \tag{2.4}$$

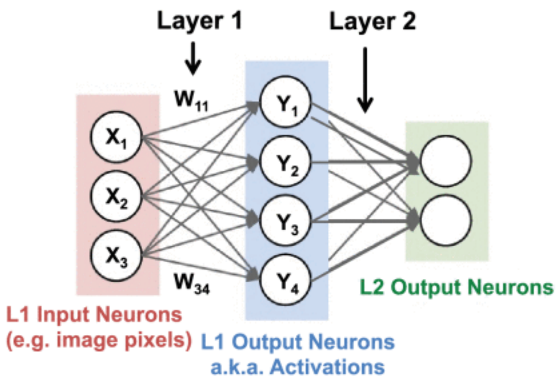


Figure 2.1: A 2-layer Neural Network (one hidden layer of four neurons and one output layer with two neurons), and three inputs [28].

where W_{ij}, x_i, y_j, and b are the weights, input data, output data and bias respectively, and f is a nonlinear function (i.e., a so called activation function) [28].

The term Deep Learning is a used whenever the networks have more than one hidden layer and are therefore called Deep Neural Networks (DNNs). DNNs are widely used in image processing tasks of the computer vision domain. A common form of DNNs used popularly to process visual data are the Convolutional Neural Networks (CNNs). Here the input could be pixels of an image with each subsequent layer responsible for extracting a particular low-level feature such as a point,lines or edges. Further layers can combine these features to eventually detect higher-level features such as shapes. Finally, given all the feature information the network can predict the likelihood that these high-level features indicate a particular object or scene.

Convolutional Neural Networks (CNNs) are stuctured in a way that preserves the spatial context of features, i.e., CNNs use a collection of pixels instead of single pixel as input to the layers of the network. These collections are trained to extract specific features from the original image and are known as convolutional filters. The output of the convolution are called feature maps. A set of such 2D feature maps form a channel. Further convolutional layers then takes the previous channel as input and perform convolution with another distinct 2D filter to obtain higher-level feature maps. The results of the convolution at each point are summed across all the channels. The final result of this computation is the output data that represent output feature map(s). CNNs can also comprise of a fully-connected layer, where all outputs are composed of a weighted sum of all inputs [9].

The fundamental component of both the convolutional and fully-connected layers are the multiply-and-accumulate (MAC) operations, which can be easily parallelized. Hence to achieve high performance there has been development of a number of highly-parallel computing paradigms and their corresponding special hardware platforms. These architectures may exploit the temporal proximity by computation on CPUs and GPUs to perform MACs in parallel or exploit spatial proximity to build energy-efficient data flow through accelerators. Such techniques have proven to reduce energy consumption and increase throughput [28].

PyTorch - a python-based library for deep learning provides all the necessary packages to implement the machine learning workflow i.e., from dataset preparation to training and testing of DNNs. The large data structures to store the inputs and parameters of the network are defined in a class called **Tensor**. These tensors can be operated on both CPUs and GPUs to accelerate computation [27].

2.3 Low-power Implementation of Deep Neural Networks

Modern Deep Neural Networks have the ability to extract key features from a large volume of raw sensor data with high accuracy. However due to their heavy computation and energy expensive tasks, model inference is usually run on cloud servers, personal computers, or smartphones. These devices have the specialized hardware with significant processing power and memory to handle the complex computations.

But the challenge arises where there is a need to perform inference in real-time and directly on-board small microcontroller units (MCUs), i.e., edge computing, where the inference is run independent of the cloud and the internet, and relying only on the capabilities of the device itself. In such applications the training process still cannot be implemented on the microcontrollers due handling of large datasets and long execution times in the training process. But an edge device should be capable of running inference on a pre-trained model onboard. For example, in tasks such as relative localization in MAV applications, safe and accurate autonomous behavior should be guaranteed despite severe constraints in payload, battery, and power consumption. Additional benefits of running inferences on-board the MCUs include reduced latency, better security, and privacy as the data stays local.

To ensure that the DNN models that are designed to maximize accuracy can be successfully implemented and deployed on the hardware designed to minimize energy, it is necessary to adopt co-designing approaches. These co-design approaches work on either reducing precision of the operands or reducing the number of operations. Some common methods are listed below [28] :

- **Quantization** involves mapping the weights and activations of the network from real numbers to integers ($\mathbb{R}$ to $\mathbb{Z}$) offering benefits like higher throughputs, lower memory requirements and compatibility with wider range of hardware as most microcontrollers in current usage support integer computation.

- **Network Pruning** is the process of removing the redundant parameters of the network and fine-tuning the remaining network to maintain accuracy. The networks are generally initialized with a higher number of parameters than required ease of training. Once the model has been trained some of the parameters may be redundant and can be set to zero. The criteria for choosing the weights that can be removed vary with application, for example removing weights that have lower impact on training loss, or weights that are lower in magnitude or even pruning weights based the energy consumption of the layers they are associated with.

- **Knowledge Distillation** involves imitating a teacher-student paradigm. Here a complex teacher model can be used to learn the properties of the dataset and then this knowledge can be transferred to a simpler student model. Thus the student model can have the same level of accuracy of the teacher while being lean and deployable. However, this method, when applied by itself, struggles to achieve the level of compression comparable to above methods without significant accuracy degradation.

Since the goal of the thesis is to be able to deploy a DNN that performs relative localization on resource-constrained hardware, both quantization and pruning methods offer significant computation reductions and memory savings necessary. While pruning reduces the number of weights and by extension computations required, keeping the remaining weights still in operation with full precision, quantization focuses on reducing the bit-width of the weights and computations in the network. Considering the features of the hardware i.e., the Crazyflie 2.1 [1] coupled with the AI deck 1.1 [2] , quantization is chosen to be the approach henceforth (see section 2.7). Quantization allows the implementation of the chosen DNN by representing the network parameters in pure integers as supported by the computing unit on board the AI deck. A recent publication comparing the two methods reveals that quantization tends to perform better than pruning without the need for changes in hardware architecture. Along with the weights and activations, pruning requires 1-bit information per weight to indicate whether this was a pruned weight or not. Additionally, to address the challenge of storing the network parameters in a compressed format, separate custom hardware units that handle the compressed parameters have been proposed. Pruning was shown to be better suited for fine-tuning networks that need to be highly compressed (2 to 3 bit width precision), which are rare in practice [11].

2.3.1 Quantization

Quantization is the process of reducing the precision of the operations and operands of the DNN thus enabling large models to be concisely stored and used for inference on devices with constraints mentioned above. Quantization involves mapping a set of real values to integer values, with the goal of having minimum error between the reconstructed data from quantization levels and the original full-precision data.

Formal definition of quantization : A tensor t is said to be quantized if all elements $t_i \in t$ can be represented as below:

$$t_i = \gamma_t + \epsilon_t \cdot q_i \tag{2.5}$$

[1]https://www.bitcraze.io/products/crazyflie-2-1/
[2]https://www.bitcraze.io/products/ai-deck/

where ϵ_t is a real valued scalar called quantum, γ_t is a real valued scalar called offset, and $q_i \in \mathbb{Z}_t$ where $\mathbb{Z}_t$ is a subset of the $\mathbb{Z}$ and is called the quantized space [5].

There are two methods of quantization, which are briefly explained below [8]:

1. Quantization Aware Training (QAT) : It is the process of quantizing a pre-trained model and then fine-tuning to recover the degradation of accuracy. Here, during every forward pass the tensors are mapped to their quantization levels but the training process uses and updates the full-precision tensors in the backward pass. The goal is to allow the network to adapt to the quantized parameters leading to better accuracy, however this process has a higher computational complexity. There is also a need for a function to estimate the gradients for the forward propagation.

2. Post-training Quantization (PTQ) : It is the process of quantizing the full-precision weights and activations of a pre-trained model without any need for re-training. The model is quantized based on the clipping and scaling ranges determined by an offline calibration process using a small subset of training data as calibration data. It is comparatively faster but has higher degradation of accuracy.

2.3.2 Ultralow-power (ULP) hardware architectures

The rising demands on the processors of the UAVs to handle complex computations on board while being energy-efficient has led to the development of ultralow-power hardware architectures. One such paradigm is the Parallel Ultra Low Power (PULP) that exploits the parallelism in vision-based CNNs without exceeding the power limitations of the UAVs. PULP [20] is a RISCV-based open-source multi-core platform designed for applications across different domains that require onboard processing on edge devices. PULP Platform also provides a PULP Microcontroller Software Interface Standard (PMSIS) that includes the Board Support Package (BSP), the Application Programming Interface (API), and the drivers for running applications on PULP-based MCUs such as the GAP8 processor. To support the deployment of complex machine learning-based applications on the small PULP processors, the PULP team provide open source tools like NEural

Minimization for pytOrch **(NEMO)**, Deployment ORiented to memorY **(DORY)**, and the Parallel ULP Neural Network library **(PULP-NN)** among others.

GAP8 : The GAP8 processor is a commercial PULP-based IoT application processor[3] with nine extended RISC-V cores. It has a fabric controller (FC) core for control, communications and security functions and 8 compute clusters (CL).

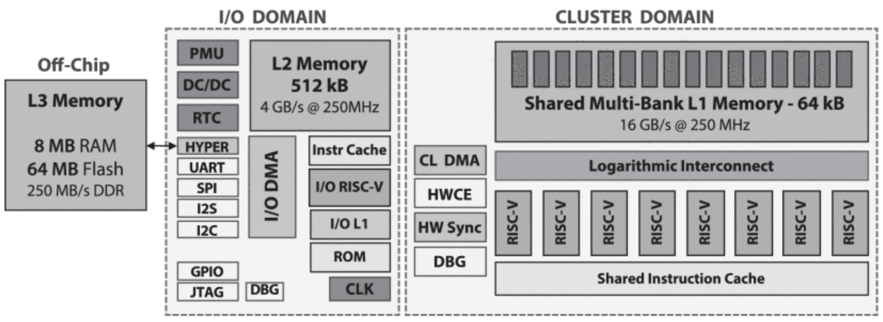


Figure 2.2: GAP-8 MCU architecture[3].

The cores in the CL share a 16 bank L1 memory (64 kB) and a 4 kB instruction cache. The FC has a 16 kB private L1 memory for data and a 1 kB instruction cache. Furthermore FC and CL share a 512 kB L2 memory for code and data. A programmable Direct Memory Access (DMA) controller handles autonomous, low power, parallel data transfers between L2 and cluster L1. The FC also has a memory protection unit to allow secure execution. The processor supports a variety of I/O peripherals like cameras and microphones and provides a number of standard serial interfaces like UART, I2C, SPI, I2S. GAP8 has been optimized to enable deployment of CNNs to low-cost, battery operated edge devices that operate on real-time sensor based data[4]. GAP8 is supplemented with a software development kit (GAP8 SDK) which includes a C/C++ compiler, tools for debugging and profiling applications and support for the PULP operating system. The GAP8 processor has a configurable frequency for the FC and CL to adjust the power consumption based on the computational workload [6].

Thus the PULP architecture and its associated tools (NEMO, DORY, PULP-NN), in combination with the GAP8 hardware and its associated SDK provide a complete

[3]https://greenwaves-technologies.com/gap8_mcu_ai/
[4]https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1_9.pdf

setup for applications demanding fast, flexible, energy-efficient, real-time and learning-based solutions. Hence for the application of enabling autonomy for small MAVs through vision-based learning methods, the PULP ecosystem seems to be a good choice to begin with. Alternate deployments flows like GAP*flow* by the manufacturers of the GAP8 SoC exist, which comprises of NNTOOL for converting a NN model into a quantized model (with PTQ) and the AutoTiler for the memory management between the different hierarchies of memory. However, the Autotiler[5] tool is proprietary unlike the PULP tools which are open-source and free for development.

2.4 NEural Minimization for pytOrch (NEMO)

NEMO is a framework for layer-wise quantization for DNN models developed using PyTorch. NEMO stages the quantization process and defines four representations of the neural network model with the objective of finally having a DNN model represented purely by integers. NEMO plays a role in training, quantization and graph optimization stages. The four representations are explained below [5]:

1. FullPrecision : This is the regular representation in which all parameters of the model are *float32* values. The model may be composed of linear operations (convolution and / or full-connected layers), batch normalization operations and non-linear activation operations. A layer is a sequence of linear operations that conclude in the first encountered Activation function.

2. FakeQuantized : This is an intermediate representation in which the functions in different layers of the *FullPrecision* model are replaced by a new quantization functions. The activations of the network are clipped to a restricted set of real values. For example, ReLU layers which in the *FullPrecision* mode clips any input given, to a value $\in [0,\infty)$, now is replaced with a clipping function that limits the activations to a value $\in [0,\beta)$. The quantum

[5]https://greenwaves-technologies.com/which-ai-model-can-run-on-the-very-edge/

ϵ, which is the smallest possible value representable, i.e., the value of 1 bit in the integer representation, is calculated at this stage as follows:

$$\epsilon = \beta/(2^Q - 1) \tag{2.6}$$

where Q is the bitwidth, usually chosen depending on the hardware used in the application.

3. QuantizedDeployable : The next stage involves completing the quantization such that the model now operates on quantized inputs and outputs quantized tensors. The weights of the linear layers are clipped to values $\in [\alpha,\beta)$ and the parameters of batch normalization are quantized. The network still operates on clipped real-valued weights but all the inputs and outputs can be decomposed into a format as defined by eq. (2.5). At this stage, the network cannot be further trained, but can be exported to an Open Neural Network Exchange (ONNX) format which encapsulates the quantization information. This stage requires the *FakeQuantized* model and the starting *quantum* ϵ calculated with eq. (2.6) as inputs. The current model, is however, still not deployable on hardwares that do not support floating-point computations, as the weights and activations are real-valued.

4. IntegerDeployable : The weights and activations of the model are now converted to an integer-only, bit-accurate representation. This process can be formalized as below:

$$\hat{t} = \epsilon_t \cdot Q_t(t) \tag{2.7}$$

where $\hat{t}$ is the quantized form of tensor t, Q_t is a quantization function that translates the real numbers to their integer representations and $Q_t(t)$ is called the **Integer Image** of t [5]. At this stage, the network can ignore the quantum and work only on the integer images of all parameters, i.e., all weights and activations are purely integers and computations are decomposed to integer multiply-and-accumulate (MAC).

The following paragraphs details some additional information important for the quantization process:

Choice of the value of Q for quantum calculation : NEMO supports mixed-precision quantization for weights and activations. With the choice of ReLU as the activation function, the tensors can be effectively implemented as unsigned integers and hence a full bit-width can be used (e.g. 16 bits, 8 bits). However due the possibility of non-negative integers, a lower precision is imposed (e.g. 15 bits, 7 bits) is imposed for the weights.

Representation of Input : To operate on an *IntegerDeployable* model with fully quantized parameters, it must be ensured that the input is also in the supported format, i.e., inputs must also have an appropriate quantized representation. For models that deal with images as inputs, it is fortunate that the inputs are naturally quantized where the image pixels are inherently represented by a value in the range of $0 - 255$.

Open Neural Network Exchange (ONNX) format[6]: ONNX is an open-source standard for representing deep learning models to enable portability of these models across different frameworks. ONNX helps decompose the model as a computational graph model with defined nodes and the connection between these nodes using a standardized set of operators and functions. It also enhances the hardware deployability of the model for real-time onboard inferences. In this use case, the PyTorch trained and NEMO quantized model can be exported in the ONNX format for deployment.

In practice, NEMO as a tool traverses the *FullPrecision* network and recognizes the super layers of the NN, i.e., typically the Convolution + Batch Normalization + Activation pattern (henceforth referred by Conv-BN-ReLU for simplicity). It then provides a linear quantization of tensors in these nodes by splitting all operations into Multiply/Addition/Shift operations for integer represention. As described by the four stages in listing 2.4, a single tensor is first translated to a fixed point tensor and then into its equivalent integer representation multiplied by a quantum. Figure 2.3) shows the decomposition of the different layers into MAC operations[7].

[6]https://onnx.ai/

[7]https://pulp-platform.org/docs/pulp_training/ABurrello_Tutorial_part1.pdf

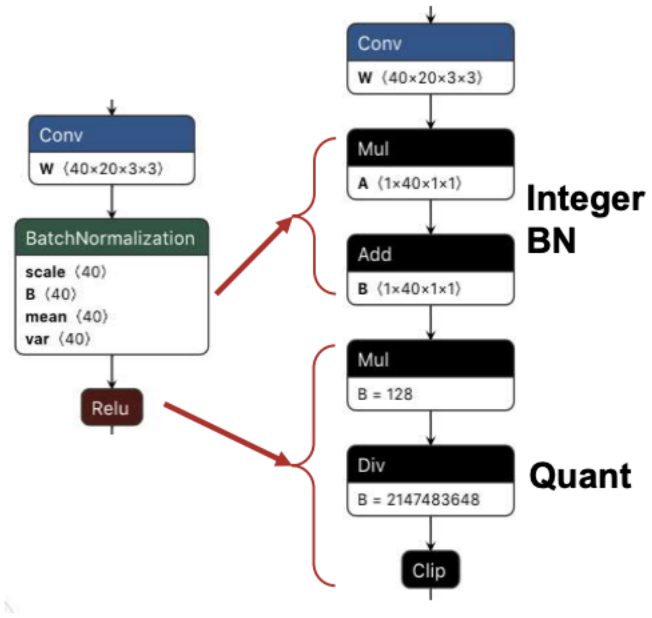


Figure 2.3: NEMO flow for quantization.

2.5 Deployment ORiented to memorY (DORY)

Operating with only standard on-chip memories (usually around 1 MB) limits the number of industrial NNs that can be supported, and the ones that can be supported too suffer from low accuracy. For example it was shown that with just 1 MB of on-chip memory, only a standard MobileNet [10] model could be supported at a poor accuracy of around 50%. But if some off-chip memory (around 64 MB) can be added and there is a mechanism to support data transfers, it can be seen that many more state-of-the-art NNs can be supported with better accuracy.

PULP-based chips have fast but very small on-chip memory (512 kB L2 memory, 64 kB L1) and off-chip memories are much bigger but not directly usable and need additional mechanisms to maintain throughput. Keeping these objectives, DORY (Deployment ORiented to memorY) - a tool that automatically manages the memory of PULP-based chips, to efficiently port DNNs on the ULP edge devices was developed. DORY plays a role in the graph optimization, memory-aware deployment and optimized DNN primitives stages.

DORY takes in an *IntegerDeployable* network (with 8-bit precision) as an input and provides a C compilable and runnable network. This is achieved by the below steps :

1. **Decoding the ONNX output**: In step 1, DORY parses the network graph in the ONNX model and looks for Conv-BN-ReLU occurences (it could also be Linear-optional BN-Activation). When a Conv/Linear layer is encountered, a new node is created. Then when the following BN recognized by the MUL-ADD operations is found, the existing node is updated with the BN node. A ReLU node is recognised by the MUL-DIV-Clip pattern in the ONNX graph and results in the update of the previous node. All Cast nodes are ignored. A MaxPool layer from the ONNX graph leads to the creation of a new node as it cannot be fused with the previous Conv-BN-ReLU node in the current implementation of DORY. Each DORY layer uses 8-bit quantized inputs, outputs, and weights, while the representation of intermediate data is 32-bit or 64-bit signed integer based on the configuration.

   The final information from this steps consists of the Layer Name, Convolution/Linear Parameters (filter dimension, stride, padding, etc.), BN and ReLU parameters, and Input and Output information to link to other nodes. The output from this stage is a new feed-forward network graph with only DORY backend compatible nodes, their dimensions, and links between nodes.

2. **Layer by Layer Tiling** : This stage deals with looking at each node, computing and allocating the memory required for the computation. Based on these requirements, the layer analyzer optimizes and generates code to run the tiling loop to orchestrate layer-wise data movement. Tiling is the act of copying a chunk (tile) of the input image and the necessary weights from the larger memory to the smaller memory for computation and storing the results back to the larger memory. While tiling helps in processing large networks, the additional movement of data between the different levels of memories causes some delays in execution. Hence there always has to be

a trade-off between the size of memory and the execution time to achieve good performance.

- L3-L2 tiling : The larger 64 MB L3 memory were introduced to support big NNs. With L3-L2 tiling, storing activations and weights in the L3 off-chip memory instead of the on-chip L2 is enabled. There is a 1D transfer of tensors from L3 (64 MB) to L2 memory (512 kB) based on the availability and location of the activations of the previous node. The tiler tries to keep output activations in L2 as much as possible to avoid redundant transfers.
- L2-L1 tiling : Almost all networks need tiling from L2 memory (512 kB) to L1 memory (64 kB). Here, 3D transfers are enabled due to dealing with smaller memory and the goal is to maximize L1 memory utilization. DORY defines this as a Constraint Programming (CP) problem and uses a solver from the open-source OR-Tools developed by Google AI12 to meet hardware and geometrical constraint. The data transfer is sped up by using double buffering i.e., copy of the next tile in parallel with computation on the current tile. The new computation uses the available output buffer while the results of the last cycle is stored in L2 and the new set of inputs are loaded into L1 concurrently.

3. **Network Parser and Generation of C-Code** : With the information about the layers from the previous stages, DORY generates a network graph with the right memory buffer sizes at each level of memory. Then every layer is converted to a function that can be called from within an application code.

Thus DORY reduces the need for manual memory management and NN optimizations by the programmer and provides a standard framework for deployment of DNNs on ultralow-power edge devices [3].

2.6 Parallel ULP Neural Network library (PULP-NN)

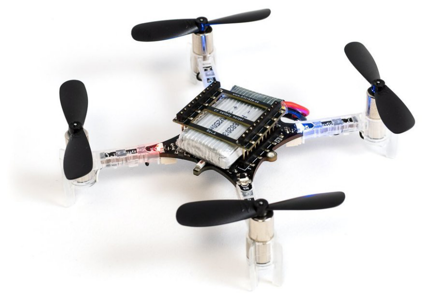
PULP-based GAP8 processor does not support floating point computation and hence better and efficient integer computation is needed. PULP-NN is an optimized backend library that mainly works on exploiting parallelism (since the processor has 8 cores present) and maximize the data reuse to accelerate computation. It operates as the last two optimization stages. The kernel uses the HWC (Height-Width-Channel) format for storing elements, and achieves the best possible data re-use by first storing pixels channel wise, followed by the width and then height dimension (for spatial locality). PULP-NN library works on the activations and weights are stored in the L1 memory [3].

2.7 UAVs and Multirotors

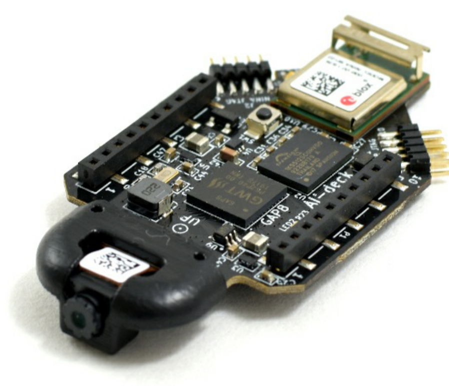
The following section describes the hardware characteristics of the UAVs used in the thesis. UAVs can be classified based on the configuration of the rotors that enable their propulsion. UAVs with three or more rotors are called multirotors. They have the ability to perform hovering and precise maneuvers and can be powered by lithium-ion batteries. Based on their size and applications there could be an overlap between the previously defined MAVs and multirotors. The implementation of the DNN for relative localization in the thesis is done on an small quadrotor platform (multirotor with four rotors) called the Crazyflie 2.1 from Bitcraze AB, coupled with an AI deck 1.1 expansion deck for complex onboard learning-based computations.

Crazyflie 2.1 is a small-sized and light-weight quadrotor designed for research. The low-level flight control is handled by a STM32F405 MCU (Cortex-M4) and the radio and power management is handled by nRF51822 MCU. It can be powered by Lithium Polymer (LiPo) battery or via a micro-USB connector. It is equipped with a host of peripherals such as the UART, SPI, I2C and additional GPIOs. The firmware to enable flight and peripherals are available as an open source repository. The Crazyflie supports hardware expansions to include other microcontroller, or add on sensors for complex applications. The onboard radio enables wireless firmware updates for ease of development. One such expansion board is the **AI deck 1.1**. The AI deck hosts a PULP-based GAP8 processor mentioned in section 2.3.2 and is equipped with a Himax HM01B0 ultralow-power 320×320 monochrome camera enabling it to run complex vision-based learning models. The Himax camera supports standard QVGA resolution (324×324) and QQVGA resolution (162×162). The desired camera resolution can be configured through a register. The AI deck has an ESP32 NINA module to provide WiFi capability for streaming images or data. It also comes with the UART peripherals to communicate with the Crazyflie host. In combination, the STM32 MCU handles the control-related tasks while the GAP8 handles the ultralow-power parallel computation.

Thus the Crazyflie extended with the GAP8 SoC on AI deck, forms a compatible platform with the PULP tools like NEMO and DORY to enable onboard inference of vision-based DNNs for relative localization of neighbors in multirotor teams.



(a) Crazyflie 2.1.



(b) AI deck 1.1.

Figure 2.4: Hardware platform.

3 Approach

The following chapter discusses the methods employed and their implementation.

A vision-based deep neural network that predicts the relative position of neighboring multirotors is quantized and deployed on a ultralow-power processor mounted on a small reseach quadrotor. Different variants of the model based on the resolution of input images are implemented with the goal of evaluating the onboard real-time performance. The goal of the evaluations is to provide a comparative analysis of the prediction quality and inference rate of the different variants. Additional analysis on the influence of the varying input resolution on the hardware requirements such as memory and computation supplement the evaluations.

3.1 Implementation Workflow

The workflow followed to implement the end-to-end deployment of a DNN that predicts the relative position of visible neighbors on-board the GAP8 processor is shown in fig. 3.1.

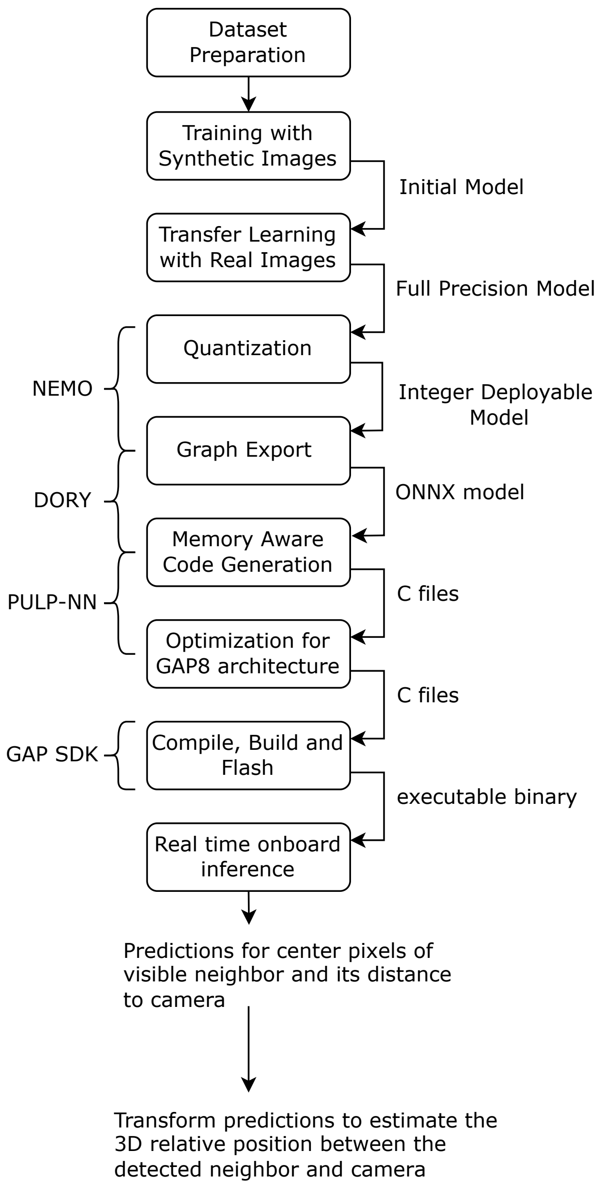

Dataset Preparation
Training with Synthetic Images
Initial Model
Transfer Learning with Real Images
Full Precision Model
Quantization
NEMO
Integer Deployable Model
Graph Export
DORY
ONNX model
Memory Aware Code Generation
C files
PULP-NN
Optimization for GAP8 architecture
C files
GAP SDK
Compile, Build and Flash
executable binary
Real time onboard inference
Predictions for center pixels of visible neighbor and its distance to camera
Transform predictions to estimate the 3D relative position between the detected neighbor and camera

Figure 3.1: Implementation Workflow.

The dataset is created from an open source large database of close-proximity flight scenario images of multirotor teams and their annotations [15]. The DNN model that predicts the center pixel of a visible neighbor and estimated distance to the neighbor from onboard camera is adapted from [12] with minor changes to the architecture (detailed below in section 3.2). The training and fine-tuning for the supervised learning of the DNN model is done using the PyTorch framework on a workstation with 32 GB RAM and an AMD Ryzen 9 3900x 12-core processor. The trained model is quantized to a model with pure integer representation of weights and activations using the NEMO library [5]. The *IntegerDeployable* network is converted to a memory-aware hardware deployable code in C network using the DORY tool [3]. The quantization and the generation of C-code for the network is done on a MacBook with the 1.4 GHz Quad-Core Intel Core i5 processor. The generated code is extended with the application code for camera operation and post processing of the network output, then compiled and built to an executable binary image using the GAP SDK[1] provided by the manufacturers of the GAP8 processors. This image can be flashed on to the AI deck microcontroller mounted on a Crazyflie 2.1 using an Olimex ARM-USB-TINY-H JTAG programmer. The network runs onboard to perform real-time inference to obtain predictions to localize the neighbor from camera images. The compile, build, flash tasks are executed on an Ubuntu 20.04 Virtual Machine on the MacBook.

[1]https://github.com/GreenWaves-Technologies/gap_sdk

3.2 DNN model architecture

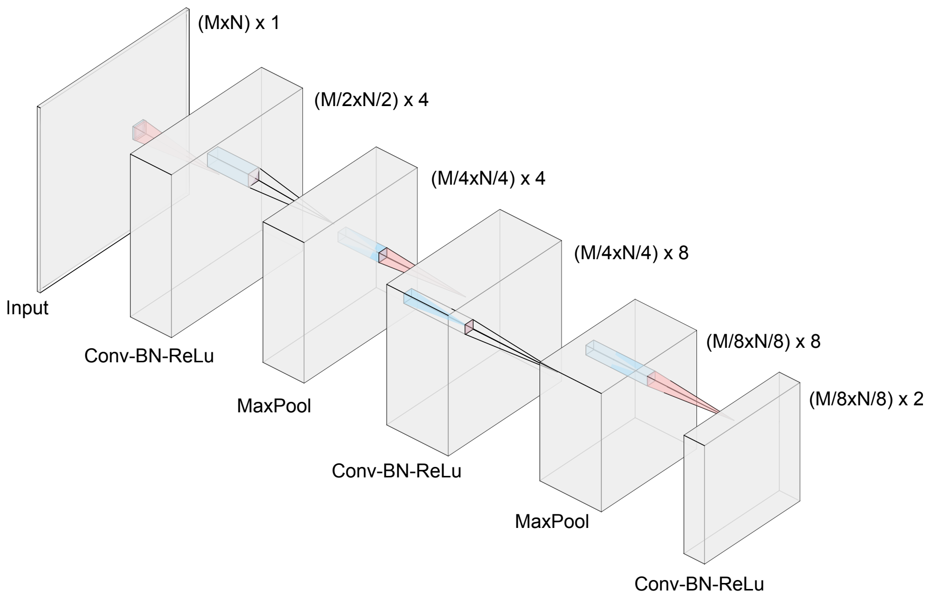


Figure 3.2: Model Architecture.

The network architecture is inspired by the work in [12] and modified in order to suit the limitations by NEMO and DORY while respecting the GAP8 memory constraints. This network architecture is similar to YOLOv3 [21] but with some changes in the output. Specifically, instead of predicting bounding boxes around the detected multirotor, the network outputs the pixel coordinates of the center of the visible neighbor and its distance to the camera. With these three parameters, and the information about the camera calibration, the 2D coordinates from a projected image can be converted back into the corresponding 3D coordinates in a three-dimensional space using inverse perspective projection (see eq. (2.3)). The input to the network is a gray-scale image of size $(M \times N) \in [320 \times 320, \quad 224 \times 224, \quad 160 \times 160, \quad 160 \times 96]$.

The layers of the network are as follows :

1. 2D Conv-BN-ReLU : The layer takes in a raw pixel values of the grayscale image of size $(M \times N)$. The convolution layer has kernel size (3×3), stride of 2, padding of 1 pixel resulting in four output feature maps.

2. 2D MaxPooling layer of kernel size (2×2) to downsample the detected features.

3. 2D Conv-BN-ReLU : The layer takes in four $(M/4 \times N/4)$ feature maps. The convolution layer has a filter of size (3×3), stride of 1, padding of 1 pixel resulting in eight output feature maps.

4. 2D MaxPooling layer of kernel size (2×2) to downsample the detection of features.

5. 2D Conv-BN-ReLU : The layer takes in eight $(M/8 \times N/8)$ feature maps. The convolution layer has a filter of size (3×3), stride of 1, padding of 1 pixel resulting in two channels - one for predicted confidence map and one for predicted distance.

Grid-based network output for relative localization: From the network grid output, the indices of the element with the maximum value in channel 1 (shown by the red dot in fig. 3.3) encode the information about the center pixel of the detected neighbor and the corresponding cell in channel 2 encodes the distance information.

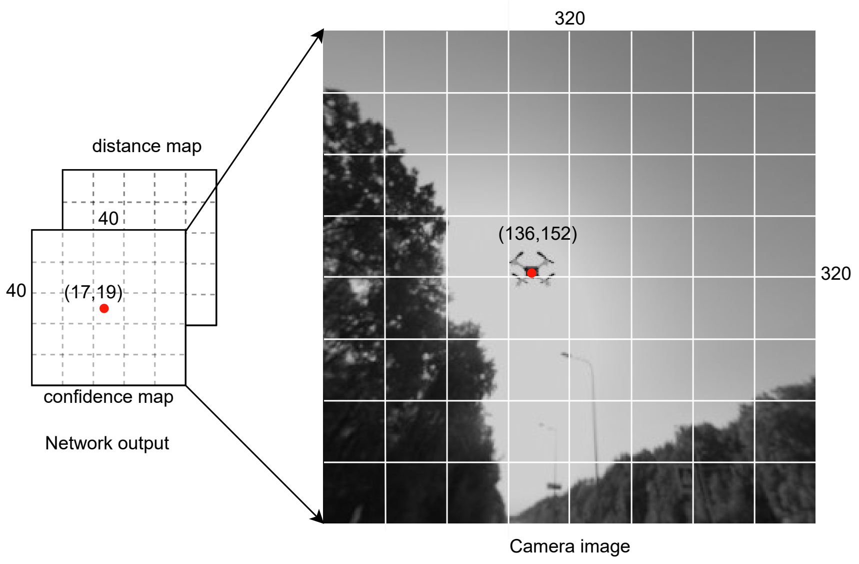


Figure 3.3: Example of the network output with reference to the camera image of size 320×320.

Design decisions made due to hardware and tool constraints:

- The first 2D convolutional layer also has the task of reducing the layer parameters such that they can be accommodated in the 512 kB L2 memory of GAP8.
- All the convolutional layers have bias turned off due to the tool limitation of NEMO in which 2D convolution biases are not quantized.
- The architecture is structured as a sequence of Conv-BN-ReLU layers as the DORY tools expects the network to be structured in this format.

3.3 Dataset and Annotations

To train the above model, the open source dataset[2] which contain images of close-proximity flight scenarios and accurate ground truth for relative position between neighboring multirotors [15] is used. This open-source dataset consists of 50000 training and 500 testing images that are 3D rendered (synthetic) and 9000 real-world images captured in an indoor flight space using a Crazyflie 2.1 with rotating single monocular camera mounted on its expansion deck. The images are gray-scale and are of size (320×320). The ground truth data provided contain are i) visible neighbor pose in world frame, ii) relative pose of visible neighbors in camera frame, iii) center pixels of visible neighbors and bounding box co-ordinates for each visible neighbor. The original 50000 synthetic training images are categorized by the number of visible neighbors leading to a smaller training dataset of 10000 training images with only one visible neighbor. The 9000 real images are also sorted for images with only one visible neighbor, leading to a smaller dataset of 1436 images. The annotations are filtered to retain only the single-robot images and the necessary information, i.e., center pixel of a visible neighbor and the distance to the visible neighbor from the camera (only the z-coordinate values of the relative pose in camera frame). For testing, the original dataset also comes with a separate 1500 image synthetic dataset independent of the training images. Similar filtering as training dataset is applied to this test

[2] `https://github.com/IMRCLab/dataset-cv-rel-pos`

images, to obtain a smaller test dataset with 500 images with only one visible neighbor and the associated annotations.

For the transfer learning phase, a subset of the real-world dataset is created with the same criteria as the synthetic dataset such that 1339 images are used for training and 97 for testing. The annotations are filtered to retain only the image names, center pixel of the visible neighbor and distance from the camera to the neighbor. Some example images are shown in fig. 3.4:

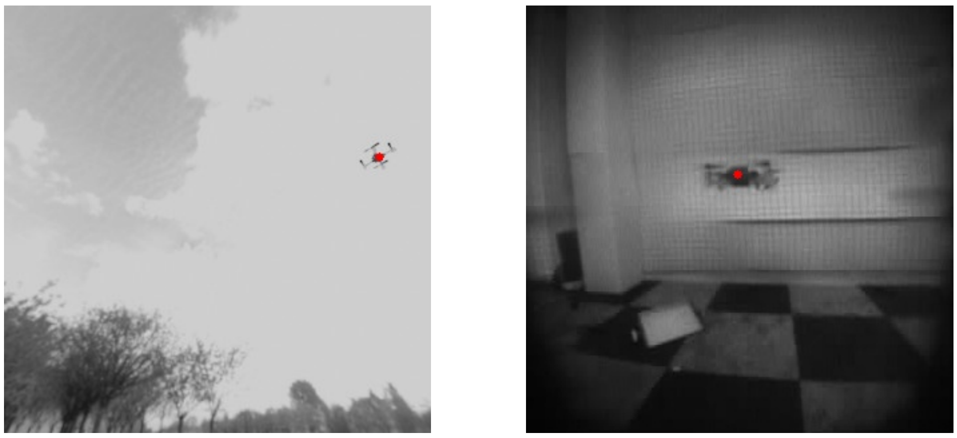


Figure 3.4: Examples from the dataset [15]. Left: Synthetic image. Right: Real image from flight experiments. The red dots mark the center pixel of the visible neighbor retrieved from the ground-truth data.

For training the model, the annotations cannot be directly used as the network output is designed to be a two channel grid. The image annotations are used to form labels that are in the $(M/8 \times N/8 \times 2)$ grid format based on the below criterion adapted from [12]:

$$\begin{cases} c(i,j) = 1, \quad d(i,j) = d_{gt}, \quad if(i,j) = (x_{gt}/8, \quad y_{gt}/8) \\ c(i,j) = 0, \quad d(i,j) = 0, \quad otherwise \end{cases} \tag{3.1}$$

where (x_{gt}, y_{gt}) represent the center pixel of the visible neighbor, d_{gt} is the distance from the camera to the visible neighbor (in meters) as calculated from the ground truth, $\mathbf{c}(i,j)$ is the value of channel 1 and $\mathbf{d}(i,j)$ is the value of channel 2 at channel indices (i,j). This means that a cell (i,j) that contains a visible multirotor has confidence $\mathbf{c}(i,j) = 1$ and the depth of $\mathbf{d}(i,j)$ in camera coordinate.

3.4 Training and Validation of the Model

The 10000 images from the above mentioned dataset are randomly shuffled and split into 8000 training and 2000 validation images. The model is then trained on the 8000 synthetic images for 50 epochs. The hyperparameters are: batch size chosen is eight and constant learning rate of 0.01. The training is run on a GPU to speed up the process. After each epoch, a validation is performed with 2000 non-training images and the loss is recorded. This is done to check the performance of the training algorithm and tune it if necessary and ensure that the model does not overfit to the training dataset. After training, the model is saved for transfer learning.

3.4.1 Loss function

The loss function is used to measure the error between predictions made by the model and the ground-truth labels and is chosen based on the application. During the training process, the parameters of the model are adjusted using an Adaptive Moment Estimation (Adam) optimizer with every iteration to minimize this loss function. The loss function used during the training of the above model is also adapted from [12]. The total loss l is defined as the sum of the confidence loss l_c and the distance loss l_d. For the confidence loss, binary cross-entropy function is used while for the distance loss, weighted mean squared error is used with a weight of 10. The weight is a fine-tuned parameter to ensure that the loss for both channels are approximately on the same scale:

$$
\begin{aligned}
l &= l_c + l_d \\
l_c &= mean\Big(\sum_{i=1}^{N/8}\sum_{j=1}^{M/8}(c-\hat{c})^2[-c\cdot\log(\hat{c})-(1-c)\cdot\log(1-\hat{c})]\Big) \\
l_d &= mean\Big(\sum_{i=1}^{N/8}\sum_{j=1}^{M/8}10\cdot c(i,j)\cdot(\hat{d}(i,j)-d(i,j))^2\Big)
\end{aligned}
\tag{3.2}
$$

where $\hat{c}(i,j), c(i,j)$ are the predicted and ground-truth confidence and $\hat{d}(i,j), d(i,j)$ are the predicted and ground-truth distance in the grid with indices (i,j) [12].

3.4.2 Transfer Learning with real-world Images

After the first phase of training, the network is refined using transfer learning as there is a difference between the synthetic images and real-world images captured by the camera onboard. The model is now initialized with the weights from the earlier phase of training and trained again for 50 epochs on a real-world dataset of 1339 images keeping the same hyperparameters and the loss function as in the previous phase. The model is saved for quantization and inference.

3.5 Quantization of the Model

The next step in the workflow is to perform quantization on the trained model. This process involves representing the model parameters, including weights and activations, as integers, making it suitable for deployment on the GAP8 processor. Post-Training Quantization (PTQ, see section 2.3.1) is performed using NEMO[3] (see section 2.4), hence there is no need to re-train the network at this stage. The quantization is performed in three stages - ***FakeQuantization*** for initial clipping of the activations, ***QuantizedDeployable*** for clipping of linear layer weights and incorporating BatchNormalization and finally ***IntegerDeployable*** to obtain a fully integer- and bit-accurate representation of the parameters. The detailed implementation of the four stages of quantization are discussed in the following section.

1. FullPrecision : The model obtained as the output of transfer learning is considered to be the *FullPrecision* model (FP) for the quantization process. This model shall also be used as a reference to check if the performance of the model is maintained throughout the different quantization stages.

2. FakeQuantized : Three tasks performed at this stage are summarized below:

   - Calculate the input quantum (see eq. (2.6)): The input quantum ϵ helps represent an input tensor with Q bits. For models dealing with image pixels

[3]`https://github.com/pulp-platform/nemo`

(integers in the range [0,255], i.e., $\beta_i = 255$) as inputs and reduced to an 8-bit representation ($Q = 8$), the calculated quantum is

$$\epsilon = \beta_i/(2^Q - 1) = 255/(2^8 - 1) = 1 \tag{3.3}$$

- Tranform to NEMO *FakeQuantized* (FQ model) : This stage takes a FP module and makes it quantization-aware, i.e., all the data structures of the convolution and activation layers of the normal PyTorch model (`torch.nn.Conv2d` and `torch.nn.ReLU`) are replaced with the NEMO versions (`nemo.quant.pact.PACT_Conv2d` and `nemo.quant.pact.PACT_Act`). The FP model is transformed into a FQ model using the NEMO library function `quantize_pact()`. The function takes in the FP model and a dummy input, which is a tensor of the same size as network input (here, a $M \times N$ tensor of random integers). Precision is set via the `change_precision()` method. Selection of the parameters for which precision should be set is based on the value of `scale_weights` and `scale_activations` flags. The value for the `bits` field is dependent on the hardware platform and on DORY constraints. In this project, an 8-bit precision for activations (non-negative integers) and 7-bit precision for weights (signed integers) is used as DORY works only with 8-bit networks. The code snippet is shown below:

```python
import nemo
import deepcopy
model_q = nemo.transform.quantize_pact(deepcopy(model_fp),
    dummy_input=dummy_input_net, remove_dropout=True)
model_q.change_precision(bits=8, scale_weights=False,
    scale_activations=True)
model_q.change_precision(bits=7, scale_weights=True,
    scale_activations=False)
```

- Offline Calibration and Quantize Activations : Here the activation layers are replaced by their quantized version, i.e., the ReLU function output is clipped to a value $\in [0,\beta)$. While the clipping parameters for weights (α,β) can be set during quantization in later stages, it is not so for clipping parameter for activations (β) which, unless explicitly modified, will be always set to a default value. Thus an offline calibration is necessary. In NEMO, calibration can be performed by running a special statistics collection mode for activations using `statistics_act()` method, which computes β by running inference over a small dataset. This is called the calibration dataset and was created with 500 randomly chosen images from the training dataset (`testing_model` refers to the inference function, `calib_loader` is the calibration dataset loader). The value of β[4] is reset to the value calculated after the satistics collection. The code snippet is shown below:

```
with model_q.statistics_act():
_ = testing_model(test_folder_path,model_q,calib_loader)
model_q.reset_alpha_act()
```

3. QuantizedDeployable (QD) : The FQ model is transformed into a QD model using the high-level `qd_stage()` method, which performs a series of steps listed below:

   - Round and harden weights of convolutional layers, i.e., freezing weights in their quantized state (the weights are limited to a value $\in [\alpha,\beta)$ but they are not yet bit-accurate to the hardware).

   - Perform quantization for the BatchNormalization layers, i.e., replacing the parameters of the BN with their quantized versions as per the rules described in [25]. The BN layers are converted to fully integer scaling operations. The precision rules for the quantization of BN parameters is not the same as in *FakeQuantized* stage but rather set to a default value within NEMO (12-bit in the current version of the tool). The BN output is now represented with 32 bits.

   - Propagating quanta ϵ along the network, for each operator. The input quantum ϵ_{in} is set explicitly to the value calculated previously in eq. (3.3) (here, $\epsilon_{in} = 1$).

```
model_q.qd_stage(eps_in=1)
```

[4]Note: Due to legacy code, in activations, the parameter referred to as β above is saved in the alpha parameter in the implementation [5], hence the method name is `reset_alpha_act()` .

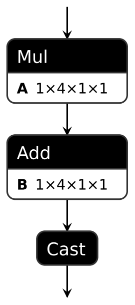
4. IntegerDeployable (ID) : The QD model is transformed into an ID model using the high-level method `id_stage()`. At this stage, the network ignores the quantum and replaces weights and activations by their integer images (see section 2.4) in all layers.

```
model_q.id_stage()
```

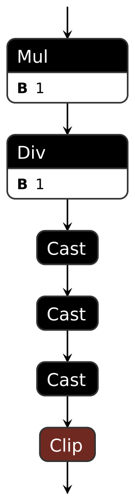
The ID model is then exported in ONNX format for usage in DORY using tools provided by NEMO as below:

```
nemo.utils.export_onnx('path/to/folder/model.onnx', model_q,
   model_q,input_size)
```

where `model_q` is the final quantized model and `input_size` is $M \times N$. The converted layers of the model as represented in the ONNX format is shown in fig. 3.5:



(a) NEMO quantized form of BN operation.



(b) NEMO quantized form of ReLU operation.

Figure 3.5: Quantized representation of BN and ReLU functions in terms of MAC operations.

The full ONNX structure can be found in the section appendix C.1. As a final step, the quantized model activations and the input pixels for one single image (so-called golden activations) from the dataset is exported as follows:

- Extract the input and output activations buffers using the `nemo.utils.get_intermediate_activations()` function. The function provides the network layer names and activations for the chosen image as a key-value pair of an ordered dictionary, one each for input and output.
- Identify the input layer by the key (here, it is the first convolutional layer) and store the associated activation values in an 1D array to a text file according to the string format specified by DORY.
- Identify the output layers by the remaining keys. As NEMO combines Conv-BN-ReLU into one single node, in this context, output layers are the ReLU and MaxPool. Then store the associated activation values in an 1D array to a text file according to the string format specified by DORY. These text files are input artifacts for DORY to generate the C-code.
- Export output quantum. The quantum value of the last layer (here ReLU) is exported using the NEMO `get_output_eps()` function. This quantum is crucial to convert the quantized network outputs back to their full-precision values. As described in section 2.4, the fully quantized ID model works only with the integer images of the real values and discards the quantum. To use the predictions of the model for further computations and actuation, the full-precision values need to be recomputed from the integer images according to eq. (2.7). The quantum for the ID models with different input resolution as calculated during the quantization is listed in table 3.1. These values vary slightly with the change in the calibration dataset.

Table 3.1: Quantum calculated for the different models.

Model Input	320×320	224×224	160×160	160×96
Quantum	0.0471	0.0321	0.0613	0.0544

3.6 Testing

After every stage of the quantization process, the model is tested with the test dataset mentioned in section 3.3 to track a potential drop in performance if any.

The output of the network is a $(M/8 \times N/8 \times 2)$ grid. channel 1 of this grid is a $(M/8 \times N/8)$ confidence map **c** and channel 2 is a $(M/8 \times N/8)$ predicted distance map **d** from which the predicted center pixel and the distance to the visible neighbor from camera can be extracted as follows:

$$(x_p, y_p) = (8 * w, 8 * h) \mid c(w,h) = \max\{c(i,j) : i,j = 1, \dots, 40\} \tag{3.4}$$

$$d_p = d(x_p, y_p) \tag{3.5}$$

where (x_p, y_p) represent the predicted center pixel of the visible neighbor, d is the predicted distance from the camera to the visible neighbor (in meters), $\mathbf{c}(i,j)$ is the value of channel 1 and $\mathbf{d}(i,j)$ is the value of channel 2 of output at channel indices (i, j). Some example predictions are shown below:

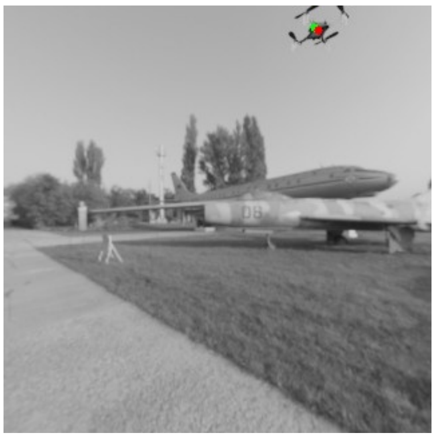
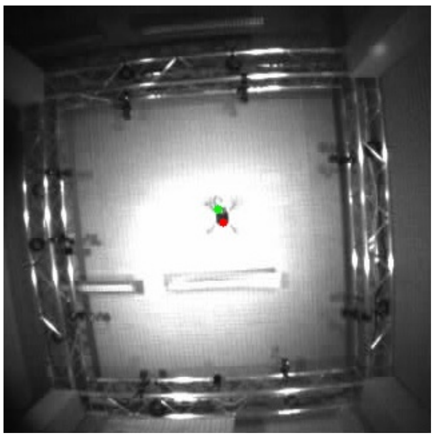



Figure 3.6: Examples from the test dataset of 320 × 320 images. Left: model predictions on synthetic image. Right: model predictions on real image..

In the above examples, a red dot is drawn over the image to indicate the center pixel of visible neighbor as taken from the annotations. The green dot indicate the pixel as predicted by the model. The values for these examples are:

- Left : Predicted Center = $(232, 16)$; Annotated Center = $(235, 18)$
- Right : Predicted Center = $(160, 152)$; Annotated Center = $(164, 162)$

Due to representation of the entire $(M \times N)$ image by a reduced $(M/8 \times N/8 \times 2)$ grid, it can be seen that the predictions are always multiples of 8 and hence sometimes a have an offset of a few pixels from the annotated values.

3.7 Deploying the Quantized Model

The quantized model can now be deployed on the resource constrained GAP8 processor for real-time inference. However the model has to be first converted to a lower-level C code that enables parallel execution over the available cores and also handles data transfers between the different levels of memory.

3.7.1 DORY

The DORY[5] tool (see section 2.5) is used to convert the quantized model into a set of C functions that perform the memory management, data transfers and network inference on the GAP8 processor.

The input required for DORY C-code generation are the ONNX model, the input and activations for one image, and a configuration file. The configuration file specifies settings for the bitwidth of the intermediate calculations, the path to the ONNX model and the allocated code space. The configurations used for this project are as below :

```
{
  "BNRelu_bits": 32,
  "onnx_file": "../outputs/checksum/testnet_ID_quant.onnx",
  "code reserved space": 297000
}
```

The DORY C-files can be generated by invoking a `network_generate.py` script provided in the tool. This code performs the following functions:

[5]https://github.com/pulp-platform/dory

1. Decode the ONNX graph of a quantized network and organize the nodes into layers of Conv-BN-ReLU sequences with quantized 8-bit integer inputs and outputs, 8-bit signed integer weights and 32-bit intermediate signed integers.

2. Determine the tiling by solving the L3-L2 tiling constraint problem (see item 2) followed by the L2-L1 while adhering to the size constraints of a network layer and relationships between the input, output and weight tensors. With L3-L2 tiling, activations and weights are stored in the L3 off-chip memory instead of the on-chip L2 memory. This enables the model with parameter sizes greater than L2 memory to be executed onboard the GAP8.

3. Generate the C code for the execution of a complete layer as per the tiling solution defined in the previous step. The data transfers are double-buffered, i.e., performed simultaneously between L3-L2 and L2-L1, and all data transfers are pipelined, thus compensating for the data transfer overhead. The pipeline is shown in fig. 3.7.

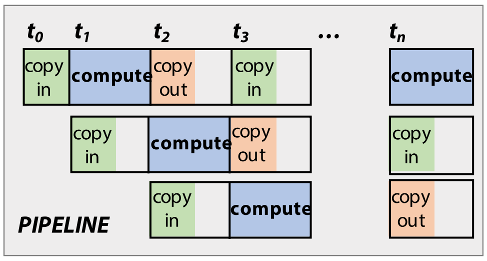


Figure 3.7: The pipeline of execution in the DORY C code. The data transfers are done parallely to the computation of the previous copied data. This is possible as the DMA calls are asynchronous and non-blocking. One call copies the weights and input activation of the next tile into L1 memory while the kernel is executed on the current tile, and the other copies the output back on the L2 memory[3].

4. Build the network with each layer as different invokable functions. The functions ensure the movement of weights from L3 to the consecutive layers of memory, and handle the execution of the layers starting from the correct memory buffers and update of the input and output buffer offsets for next execution.

The different generated files are described below:

1. Binary files : The weights, activations and input data of the single image exported by NEMO is converted to layer-wise .hex files that can be loaded on to the L3 memory. These files help in performing the checksum tests to ensure that the quantized network is running correctly on the hardware.

2. .c and .h files : These contain the functions implementing the different tasks involved in running the onboard inference. The `main.c` file holds the start of execution, functions for peripheral tasks (such as camera operations, UART for communication with peripherals, memory initialization), and running the network. The `BNReluConvolution*.c, Pooling*.c` files describe the transfer of tiles containing the network parameters from L3 to L2 and transfer of output from L1 to L2 for the respective network layers. `pulp_nn_*.c, pulp_nn_*.h` are a set of files executing the computational backend PULP-NN library, that handles parallelized computation on L1 data over the eight cores. `network.c, network.h` are files that contain the layer-wise size information, handle allocation of buffers for parameters needed by the network execution and triggers initialization, checksum tests, running and termination of the network. It also contains some functions for performance analysis.

The network summary for the largest model as calculated by DORY in the `network.h` is shown:

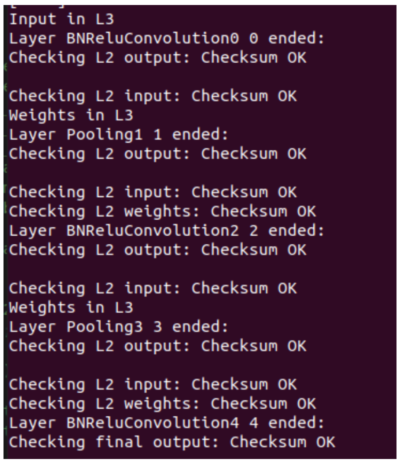
```
static const char * L$3$_weights_files[] = {
  "BNReluConvolution0_weights.hex", "BNReluConvolution2_weights.
    hex", "BNReluConvolution4_weights.hex"
};
static int L3_weights_size[3];
static int layers_pointers[5];
static char * Layers_name[5] = {"BNReluConvolution0", "Pooling1",
     "BNReluConvolution2", "Pooling3", "BNReluConvolution4"};
static int L3_input_layers[5] = {1,0, 0, 0, 0};
static int L3_output_layers[5] = {0, 0, 0, 0, 0};
static int allocate_layer[5] = {1, 0, 1, 0, 1};
static int branch_input[5] = {0, 0, 0, 0, 0};
static int branch_output[5] = {0, 0, 0, 0, 0};
static int branch_change[5] = {0, 0, 0, 0, 0};
static int weights_checksum[5] = {5399, 0, 43742, 0, 9355};
```

```
14 static int weights_size[5] = {68, 0, 352, 0, 80};
15 static int activations_checksum[5][1] =
       {{3176002},{63118},{175245},{117316},{39174}};
16 static int activations_size[5] = {102400, 102400, 25600, 51200,
       12800};
17 static int out_mult_vector[5] = {1, 1, 1, 1, 1};
18 static int out_shift_vector[5] = {23, 0, 22, 0, 22};
19 static int activations_out_checksum[5][1] =
       {{631186},{175245},{117316},{39174},{4}};
20 static int activations_out_size[5] = {102400, 25600, 51200,
       12800, 1600};
21 static int layer_with_weights[5] = {1, 0, 1, 0, 1};
```

3.7.2 Memory Management and Checksum Tests

An important consideration to be made during the deployment of the model for inference is related to the memory management for the application code and the weights and activations of the network. The largest number of activations exists for the model with the highest input image resolution where the first and second layer occupy 102400 bytes (line 16 in the network summary code 3.7.1) each. The outputs are stored in 3200 bytes (which correspond to the $40 \times 40 \times 2$ grid). During the execution of the network, the inputs are obtained from a separate L2 buffer. This L2 buffer is the camera buffer that gets populated with the pixels of the image captured by the camera. Since the largest network works with images of size 320×320, the L2 camera buffer must be allocated at least with 102400 bytes for the image pixels. Thus adding up the above mentioned partial memory requirements, the L2 buffer should be greater than $102400 + 102400 + 3200 = 208000$ bytes. Additionally smaller memory requirements for weights and intermediate results should be considered. Thus the choice made for code space is 297000 bytes leaving 215000 bytes of the 512000 bytes of L2 memory for the weights and activations. Even though the DORY policy enables execution of layers whose parameters are larger than 512000 bytes of L2 memory, experiments in [3] show that having activations larger than the L2 memory bounds and loading them from L3 slows down the execution of the network. Hence the network is constructed in such a way, that the layer with largest number of activations can still fit within the L2 memory bounds.

To guarantee that the DORY C-code deployed on the GAP8 is accurate and that the memory constraints are met, checksum tests are done, i.e., checksums of the weights and activations of every layer on the GAP8 are compared to those of the golden activations. The checksum results for all four models are ensured to be 'OK' before proceeding to the experiments:



```
Input in L3
Layer BNReluConvolution0 0 ended:
Checking L2 output: Checksum OK

Checking L2 input: Checksum OK
Weights in L3
Layer Pooling1 1 ended:
Checking L2 output: Checksum OK

Checking L2 input: Checksum OK
Checking L2 weights: Checksum OK
Layer BNReluConvolution2 2 ended:
Checking L2 output: Checksum OK

Checking L2 input: Checksum OK
Weights in L3
Layer Pooling3 3 ended:
Checking L2 output: Checksum OK

Checking L2 input: Checksum OK
Checking L2 weights: Checksum OK
Layer BNReluConvolution4 4 ended:
Checking final output: Checksum OK
```

Figure 3.8: Checksum tests for the model onboard GAP8.

3.8 Camera Calibration

The final network output is used to estimate the 3D relative position of the detected neighbors using eq. (2.3). In addition to the network output, a camera intrinsic matrix is required for the relative localization. A camera calibration process to obtain the intrinsic parameters is carried out with a known checkerboard pattern. A series of images of the checkerboard pattern is captured from the camera at various positions and orientations. Then the camera intrinsic parameters are computed with the help of the open source scripts from the IMRC lab[6] [15]. This process is performed for calibration images of sizes 320×320, 224×224, 160×160 and 160×96.

[6]https://github.com/IMRCLab/dataset-cv-rel-pos

3.9 Develop Application C Code

The generated DORY C files have a standard code that performs loading of the golden activations, allocate L2 memory for the network (here 215000 bytes) and run the checksum tests. For the application in this project, additional functionalities must be added for vision-based state estimation onboard the chosen ultralow-power hardware platform (see fig. 2.4). The functionalities are developed using a board support package (BSP) provided by the manufacturers of GAP8. The flow of execution is as follows:

- Initialize : Set the camera configuration format (see section 2.7) and camera parameters such as auto exposure gain, digital gain and exposure based on the lighting conditions of the test environment. Also allocate input buffers for the camera image pixels and output buffers for the network predictions. The chip frequency is set to 100 MHz for both the FC and CL. The voltage is set by default to 1 V.

- Camera operation and image processing : Open the Himax monochrome camera, capture an image, and close the camera. The captured image is of size 324×324 with two rows of dead pixels on all sides. The captured image must be cropped to remove the dead pixels to obtain the appropriate image size to be used as an input to the network.

- Network inference and post processing : Once the image is cropped, it is sent to the network for inference using the generated functions from DORY. The prediction output of the network is a 2-channel grid output stored as long 1D array. The center pixel is calculated from the prediction using eq. (3.4) and distance to camera is calculated using eq. (3.5). Post-processing also includes dequantization as necessary. For channel 1, the outputs (i.e., the center pixel of the neighbor) are naturally integers hence there is no conversion required. However, for channel 2, the integer image of the distance prediction should be dequantized to full-precision as follows:

$$d = d_p * NEMO\_QUANTUM \tag{3.6}$$

where d_p is the distance prediction obtained from the network output and $NEMO\_QUANTUM$ is a constant real number obtained from the quantization process as described in section 3.5. The quantum used for the ID models with different input resolution is taken from table 3.1.

- Timing calculations : Two timestamp measurements are taken - one before the opening of the camera and one after the prediction post processing. The difference in these two values give the inference time for one image.

4 Results

The following chapter describes the experimental methods and the results obtained for benchmarking the onboard inference of the quantized model in comparison to the original full-precision model. First, a comparison between the FullPrecision (FP) and IntegerDeployable (ID) models with four different input image resolutions (320×320, 224×224, 160×160 and 160×96) is done. The comparisons evaluate predictions of the two output parameters, i.e., center pixel of the visible neighbor and distance to it from the camera on the test dataset. These tests are run on a laptop with the i5 processor.

Subsequently, a series of experiments are performed after deploying the quantized models on the GAP8 processor. The onboard inference results are compared against the FP model predictions for the two output parameters as before. The predictions are then used for computing the relative position of the visible neighbor with respect to the camera.

A comparative analysis is carried out to assess the onboard performance of the models under various input resolutions and the trade-offs that have to be considered for prediction quality and the inference speed (frames/s). Finally, the different model variants are evaluated for their prediction quality against the ground-truth data using 97 test images from the real image dataset of [15]. The summary of the network characteristics that affect the onboard inference for the four quantized models are shown in table 4.1.

Table 4.1: Summary of the model characteristics for one frame inference.

Model Input	320×320	224×224	160×160	160×96
Min. L2 Memory required [kB]	400.58	196.58	100.58	60.58
Parameters ($\times 10^3$)	294.980	144.836	74.18	44.74
Operations [MAC]($\times 10^6$)	2.88	1.47	0.75	0.45

The minimum L2 requirements are based on the analysis done in 3.7.2 and considers the sizes of the weights, activations, input and output. The parameters, refer to the total number of weights and activations in the network. Operations, refer to the number of multiply-and-accumulate (MAC) computations needed to perform one input image inference.

4.1 Comparison of the FP model with the ID model

The training of the network is performed on a Ubuntu workstation equipped with a GPU for 50 epochs. The quantization is performed on an Intel *i*5 MacBook laptop. After each stage of quantization, the predictions for center pixel of visible neighbor and its distance to camera are stored in separate `yaml` format files to be used for comparison (for stagewise comparision of the quantized model, see A).

The accuracy drop due to quantization, occur in the first stage due to clipping of the weights and activations to a fixed-point threshold during FQ stage. Specifically, the performance of the FP model against the ID model on the laptop is observed for all images in the test dataset, i.e., 500 synthetic images and 97 real images that are unseen during training. The error is computed as the difference between the predictions of the FP model and the dequantized predictions using eq. (3.6) of the ID model for both center pixel (x_p,y_p) calculated from eq. (3.4) and distance d calculated from eq. (3.5).

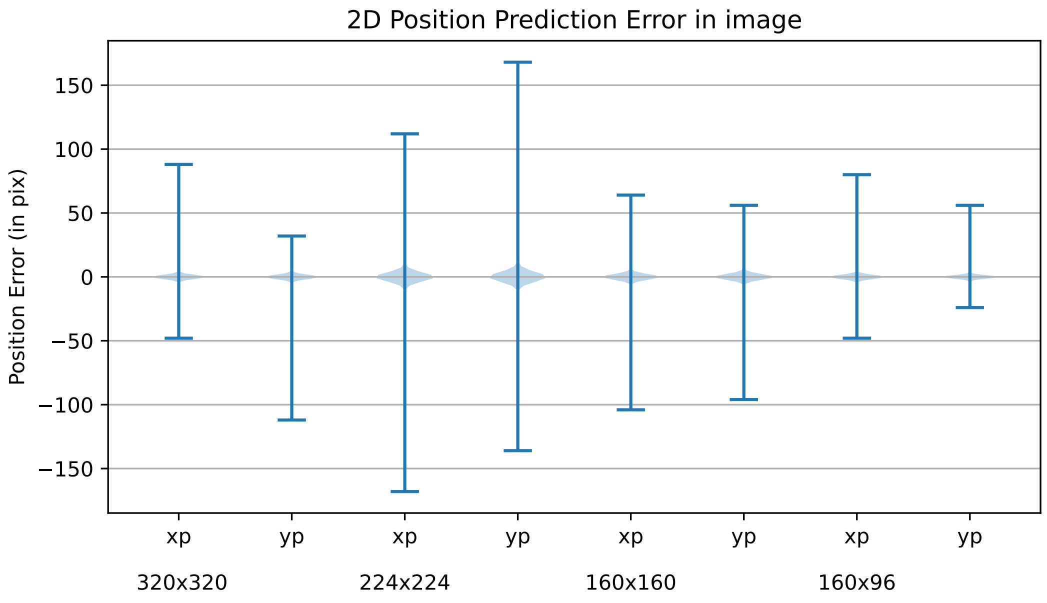


Figure 4.1: Prediction errors for center pixel of visible neighbor between the FP and ID models on 500 synthetic test dataset.

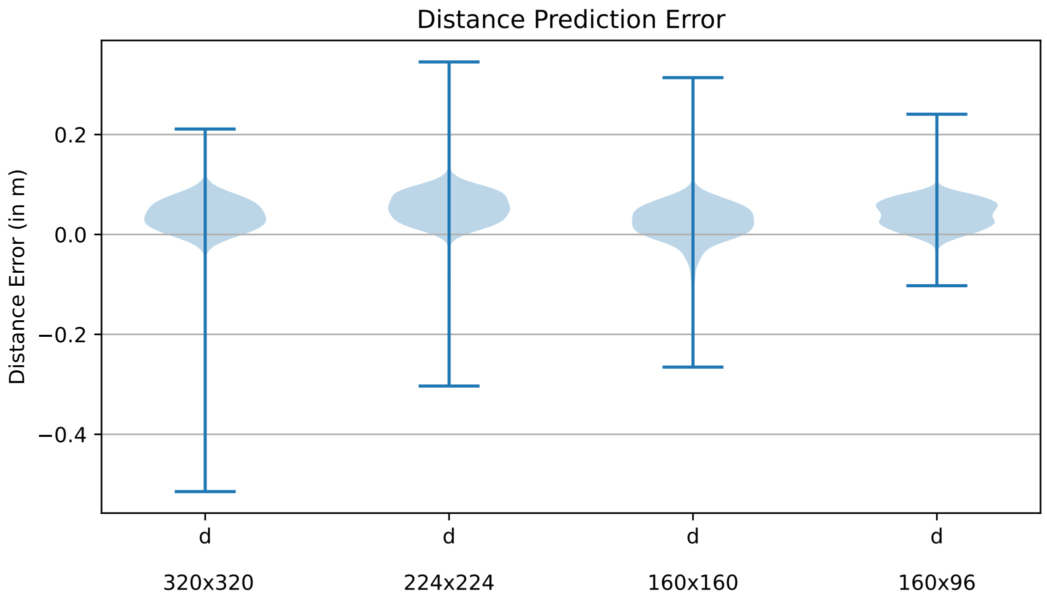


Figure 4.2: Prediction errors for distance to visible neighbor between the FP and ID models on 500 synthetic images test dataset.

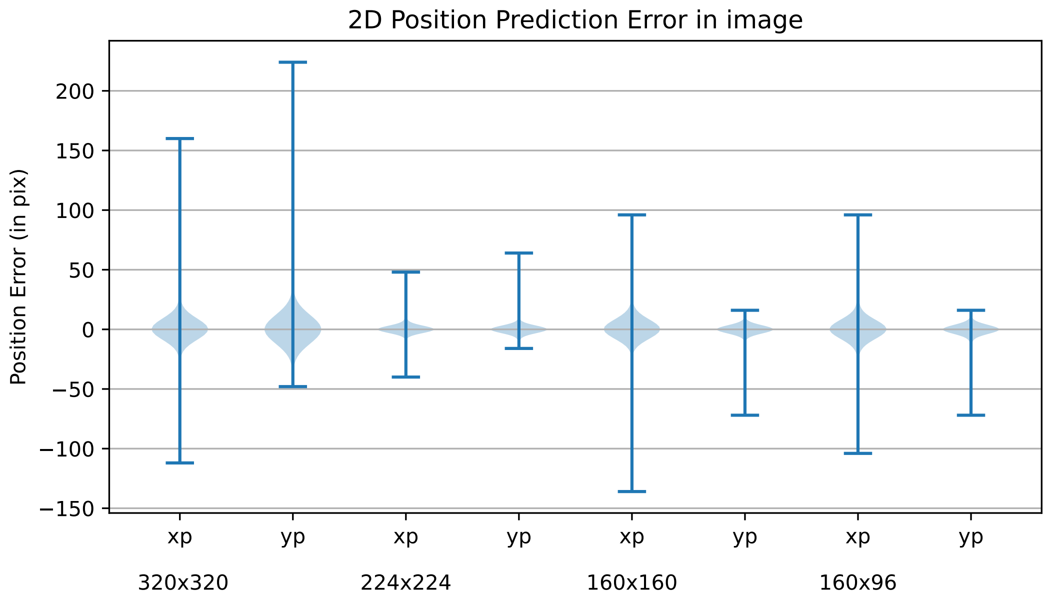


Figure 4.3: Prediction errors for center pixel of visible neighbor between the FP and ID models on 97 real test dataset.

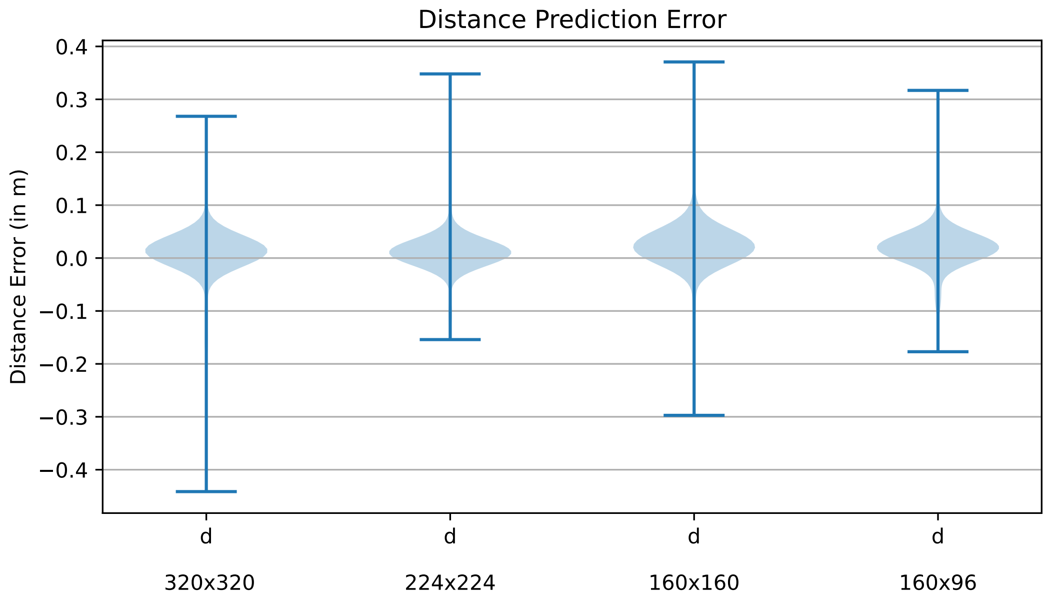


Figure 4.4: Prediction errors for distance to visible neighbor between the FP and ID models on 97 real images test dataset.

4.2 On board inference on a robotic platform

The robotic platform used for the onboard inference of the DNN model for vision based relative state estimation is Bitcraze Crazyflie 2.1 quadrotor extended with the AI deck 1.1. It has the computational capabilities to run complex DNNs onboard (GAP8 processor) and other supporting peripheral such as a the Himax camera module, and an ESP32 module for WiFi connectivity for streaming images and data.

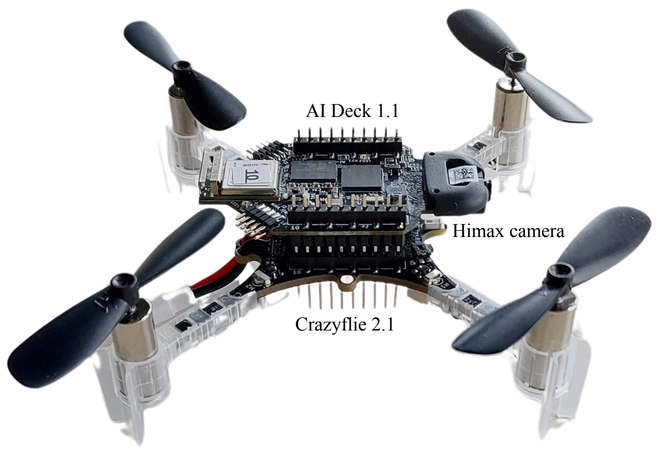


Figure 4.5: Crazyflie 2.1 with mounted AI deck 1.1 for experiments.

4.2.1 Setup of Crazyflie 2.1 and AI deck 1.1

The Crazyflie 2.1 is assembled following the instructions on the Bitcraze website [1]. It is powered by a small 3.7 V LiPo battery. Upon the powering on the Crazyflie, it automatically runs through a short sequence of self tests and sensor calibrations to get ready to fly. Bitcraze also provide a Crazyradio 2.0 USB radio dongle and the Crazyflie client software with a GUI that can be used for controlling the Crazyflie, flashing firmware, setting parameters and logging data. The Crazyflie device address is changed from the default (`radio://0/80/2M/E7E7E7E7E7`) for unique identification through the Crazyflie client. To enable the Crazyflie to fly,

[1] https://www.bitcraze.io/documentation/tutorials/getting-started-with-crazyflie-2-x/

communicate with the control software, and interact with sensors and peripherals onboard, Bitcraze's open-source firmware [2] is used. The firmware is customized to change the WiFi configurations for the expansion AI deck from default values to the right network ID and password [3]. The firmware can then be compiled and built using the RISC-V C/C++ tool chain with GNU Compiler Collection (GCC) on an Ubuntu 20.02 Virtual Machine. The new binary is flashed wirelessly via the Crazyradio [4].

The AI deck 1.1 is an expansion deck that can be directly mounted on top of the Crazyflie and is also powered by the battery connected to the Crazyflie. The Bitcraze tutorial [5] for working with AI deck was followed to flash the AI deck firmware (both ESP32 and the GAP8) also wirelessly via the Crazyradio.

For the application development, i.e., onboard inference of DNN on the GAP8, an environment with the GAP SDK installed is required. Fortunately, Bitcraze also provide a Docker container which has all the necessary configurations required for building and flashing the custom application code (here, the extended DORY C code). The generated DORY C code also includes a configurable Makefile for providing the settings for compilation. Here, we proceed with the default settings and modify only the number of cores in operation through the variable `CORES` for the experiments.

4.2.2 Experimental Setup

The first step consists of data collection, where the AI deck mounted on top of the Crazyflie (CF1) is flashed with an application code that can continuously take images from the camera and stream them simultaneously to a device connected on the same WiFi network. The camera is configured for full resolution readout by setting the register `0x3010` to 1 (`PI_CAMERA_QVGA`). Then a different Crazyflie (CF2) is placed on a surface and images of this stationary CF2 are taken from all

[2] `https://github.com/bitcraze/crazyflie-firmware`

[3] `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/development/kbuild/`

[4] `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/building-and-flashing/build/`

[5] `https://www.bitcraze.io/documentation/tutorials/getting-started-with-aideck/`

different angles from the camera onboard CF1 with the WiFi image streamer application[6]. The pictures captured are of size 324 × 324 of which four border pixels are discarded on all sides for an effective resolution of 320 × 320, as described in section 3.9 . 100 images are taken and saved for all further tests. The same 100 images are resized and saved separately for each of the smaller resolutions (224 × 224, 160 × 160, 160 × 96) Some images from the dataset are shown below:

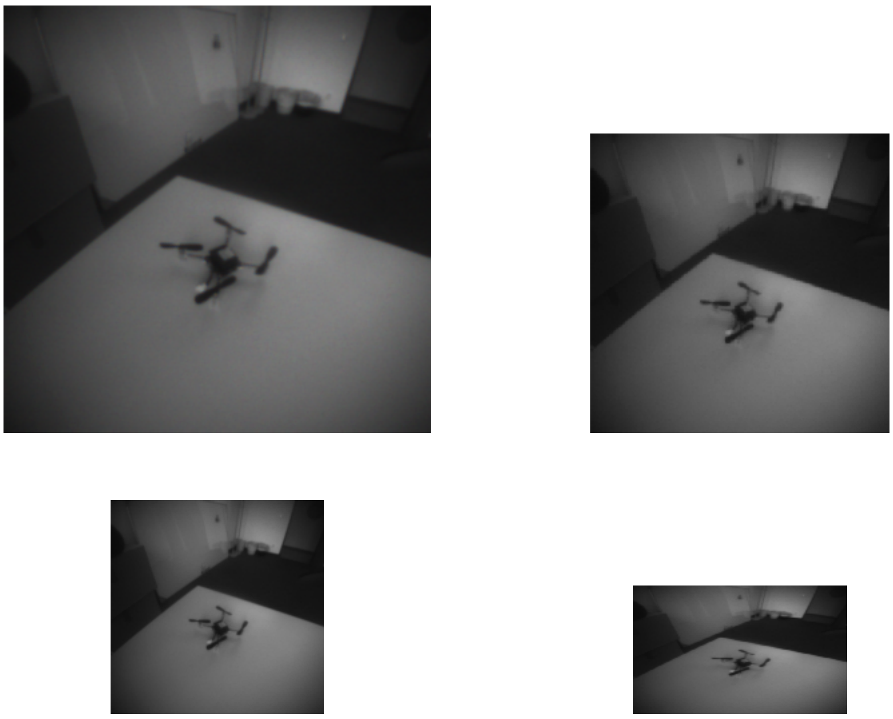


Figure 4.7: Sample from the images collected with CF1. Top Left : 320 × 320, Top Right : 224 × 224, Bottom Left : 160 × 160, Bottom Right : 160 × 96.

4.2.3 Onboard inference of 100 recorded real-world images

On the laptop, inference is run on the recorded and resized 100 images through both the FP and ID models for each of the input resolutions. The predictions from these models are set as a benchmark to compare the performance on the GAP8 processor. To test the performance of the quantized model on the GAP8 processor,

[6]`https://github.com/IMRCLab/aideck-gap8-examples/tree/cvmrs/examples/other/wifi-img-streamer`

the application code generated by DORY is extended to load an input hex file (representing a single image) from the external L3 memory into an L2 buffer, run inference on the image, and compute the network output. This application is built and then flashed to the AI deck 1.1 using the Olimex ARM-USB-TINY-H JTAG programmer. The decision to use a wired connection for flashing was made due to the binary file's size, which made wireless flashing less efficient. The 100 images used for FP and ID model bench-marking are then sent for inference (one by one) on the GAP8. The predictions from the network are post processed according to eq. (3.4) and eq. (3.5) to obtain the center pixel of the detected Crazyflie (CF2) and the distance to it from the camera of CF1. The output from the AI deck was logged to a `.txt` file via the Olimex ARM-USB-TINY-H JTAG debugger and subsequently parsed to extract relevant results.

The inference results of the three models (FP, ID and GAP8) on these 100 images are stored in separate `yaml` files and compared. The comparison is a simple error computation (FP vs ID and ID vs GAP8) for the three outputs from inference on the model running on CF1, i.e.,

1. difference between the predictions of x-coordinates of the center pixel in image of CF2
2. difference between the predictions of y-coordinates of the center pixel in image of CF2
3. difference between the distance predictions to CF2

Notably, for all of the 100 images, the error between the ID models and the GAP8 models was consistently zero. Thus, both the distance predictions and the CF2 center pixel predictions are identical for the ID model running on the laptop and the GAP8 model executing on the AI deck. This check ensures that the memory management on the GAP8 does not violate the bounds and the x_p, y_p and d computations performed onboard is consistent with the computations done in the comparison script on the laptop.

The errors in inference on the same 100 images (`CORES` = 8) between the FP model and the model running onboard the AI deck are shown in the plots below:

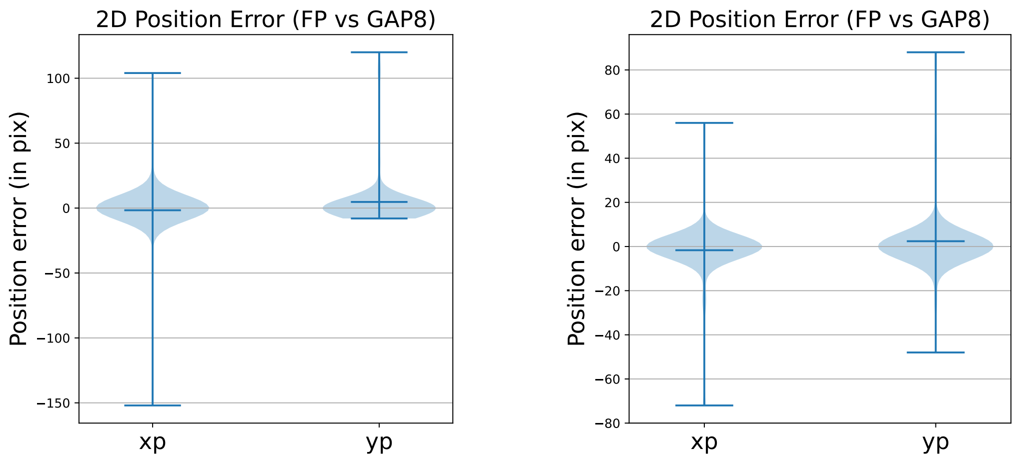


Figure 4.8: Prediction Errors between FP and GAP8 for Left : 320 × 320 and Right : 224 × 224.

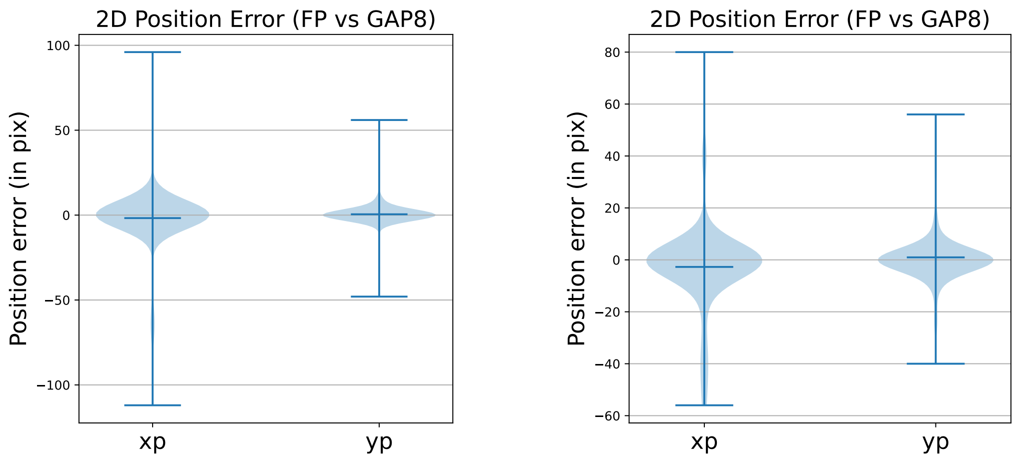


Figure 4.9: Prediction Errors between FP and GAP8 for Left : 160 × 160 and Right : 160 × 96.

There is a slight drop in performance of the quantized model running onboard the GAP8 processor as compared to the FP model. Most CF2 center pixel predictions are within ±32 pixels for the larger resolutions and within ±8 pixels for the smaller resolutions with some outliers due to wrong predictions. While inspecting the predictions, it was noticed that the higher resolution models with 10 wrong predictions each performed better than the lower resolution models with 16 and 27 wrong predictions for (160 × 160) and (160 × 96) images respectively.

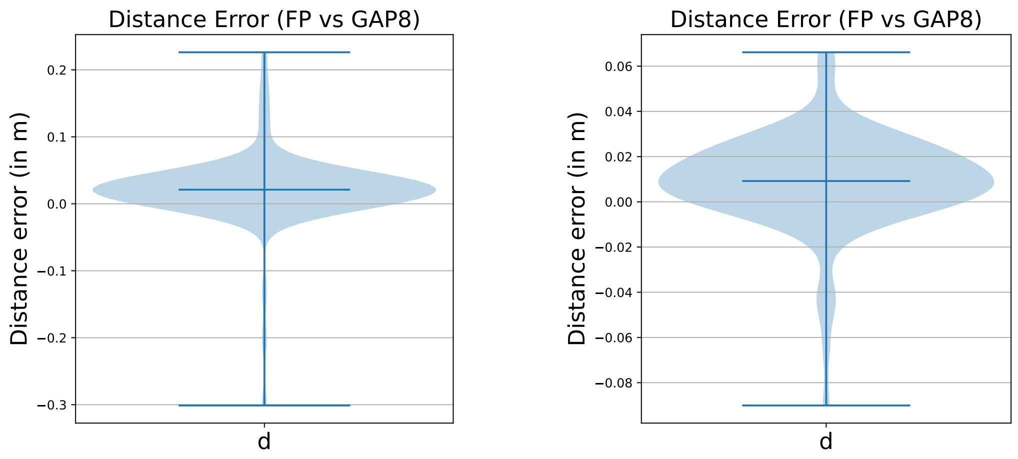


Figure 4.10: Prediction Errors between FP and GAP8 for 320 × 320 (Left) and 224 × 224 (Right).

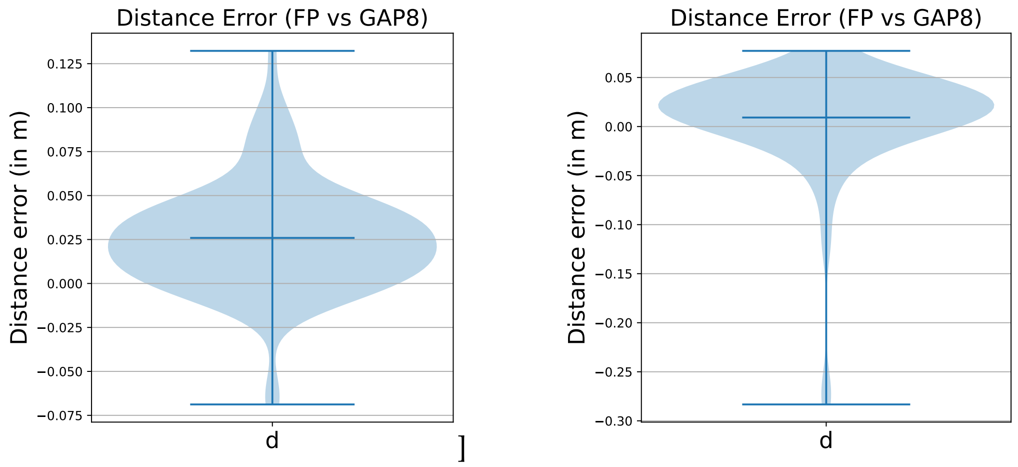


Figure 4.11: Prediction Errors between FP and GAP8 for 160 × 160 (Left) and 160 × 96 (Right).

On inspecting the predictions for the distance to the center of CF2, it was observed that the distance errors had a maximum mean of 0.025 m for the 160 × 160 variant and the model that best maintained its performance onboard was the 224 × 224 variant with a mean error of 0.0092 m. The predictions for pixel coordinates from the three different models are visually annotated on the respective images. Additionally, the error in distance predictions between the FP and ID models is superimposed and displayed on the image. Some visual representations of the predictions are shown below:

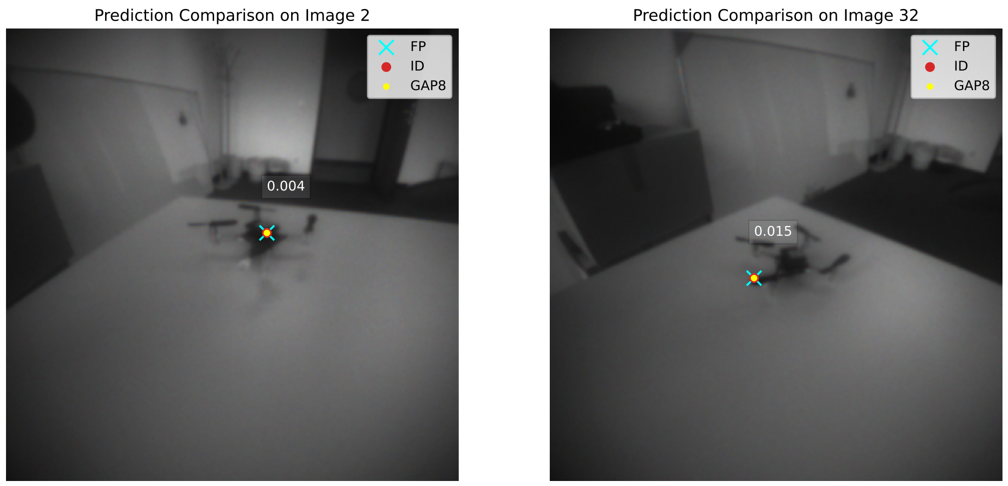

Prediction Comparison on Image 2
FP
ID
GAP8
0.004
Prediction Comparison on Image 32
FP
ID
GAP8
0.015

Figure 4.12: Left: Predictions close to the actual center of CF2 and consistent predictions between the three models with 0.4cm error in distance prediction. Right: Predictions on the propeller of CF2, slightly offset from center but consistent predictions between the three models. Higher error of 1.5cm in distance prediction (Image size = 320 × 320).

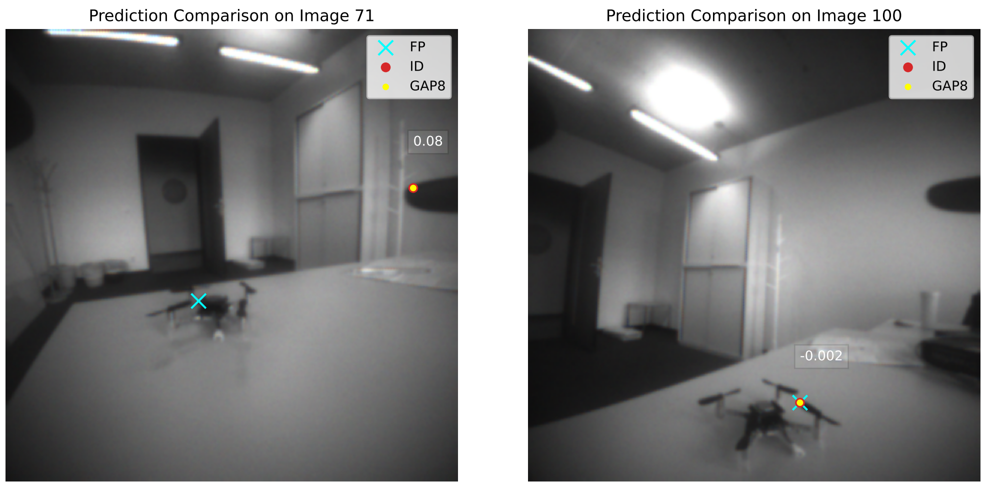

Prediction Comparison on Image 71
FP
ID
GAP8
0.08
Prediction Comparison on Image 100
FP
ID
GAP8
-0.002

Figure 4.13: Left: False predictions of ID and GAP8, center predicted to be on the CF1 propeller visible in the image. Right: False predictions on the surface close to the propeller of CF2, but consistent predictions between the three models. Error of 0.2cm in distance prediction (Image size = 320 × 320).

The same process as above is repeated to observe the performance of the model as the number of operational cores of the GAP8 processor is limited. Inference for 100 images is done for `CORES` = 1, 2, 4 and 8. As the binary file that runs onboard remains the same, it was observed that the predictions remained consistent for all resolutions. There were no additional reductions in the quality of predictions when the number of cores was limited.

4.2.4 Inference speed with hardware configuration

In the second set of experiments, the application code is extended to also include camera operation and time measurements as described in section 3.9. The application code is allowed to run continuously to capture a new image, crop the dead pixels, send image for inference and compute the outputs, repeated for 100 instances. The time for inference on one image (one frame) is calculated as an average of the 100 measured inference times. The experiment is repeated by changing the number of cores in operation to $1, 2, 4$, and 8. The observations made for the inference time per image for the different number of operational cores is as below:

Table 4.2: Single frame inference time (in ms) vs Cores for different resolutions.

Input Resolution	Cores			
	1	2	4	8
320×320	220.61	155.74	123.51	107.48
224×224	143.40	111.31	95.50	87.71
160×160	95.31	78.86	70.68	66.73
160×96	79.14	69.24	64.34	62.10

Reducing the number of active cores in GAP8 slows down the execution of the network as the number of parallel computations that can be performed is limited. The table 4.2 lists the inference times measured for the predictions on one single frame as the number of active cores are varied. This information can be used to calculate the achievable inference speed in frames/s :

$$\text{Inference Speed (frames/s)} = 1 \div \text{Inference time for a single image (s)} \tag{4.1}$$

The graph below shows the achievable inference speed as a function of the number of active cores for the different model variants. If the input resolution is reduced, and the GAP8 processor is set to utilize all 8 cores for computations, then the reduced model achieves a higher inference rate than the original model.

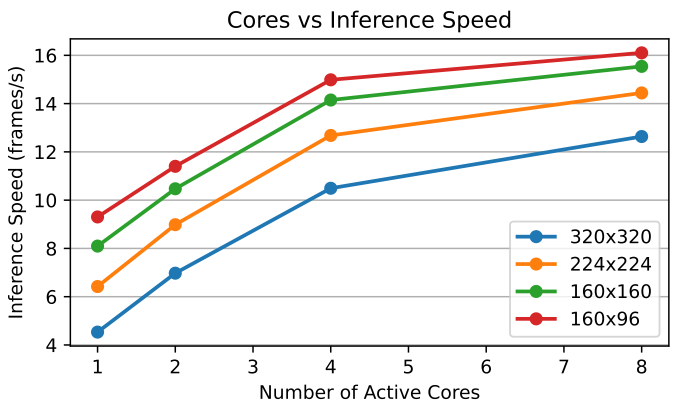


Figure 4.14: Variation in inference speed with respect to change in active cores.

4.2.5 Prediction quality with quantization

The predictions from all models is used to compute the 3D position of the visible neighbor (CF2) relative to the camera onboard CF1 using the equations described in eq. (2.3). The intrinsic parameters of the onboard camera of CF1 is obtained from the camera calibration process described in section 3.8 . To observe how the false predictions of the quantized model translate to the errors in the 3D relative position, Euclidean distance metric is used:

$$Error = \sqrt{(x_{FP} - x_{GAP8})^2 + (y_{FP} - y_{GAP8})^2 + (z_{FP} - z_{GAP8})^2}, \qquad (4.2)$$

where (x_{FP},y_{FP}, z_{FP}) and (x_{GAP8},y_{GAP8},z_{GAP8}) refer to the 3D position of the detected neighbor computed from the predictions of the FP model and from the GAP8 respectively. The error between the 3D relative position estimation computed from eq. (4.2) is shown below :

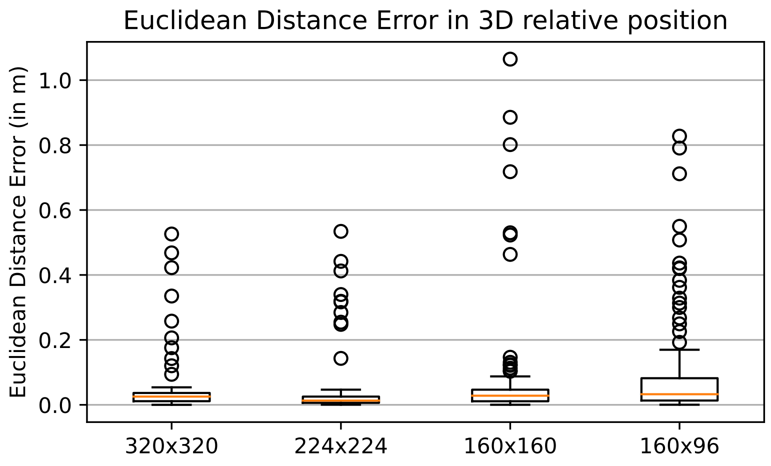


Figure 4.15: Euclidean Distance error in the 3D relative positions estimated for 100 images.

From the observations, it can be seen that the model that best maintained its onboard performance to the FP model was the one with input images of size 224 × 224. The mean and standard deviations of the error for the 100 images recorded from Crazyflie is reported in table 4.3:

	Euclidean Error	
Input Resolution	μ	σ
320 × 320	0.047	0.09
224 × 224	**0.046**	0.101
160 × 160	0.103	0.17
160 × 96	0.08	0.184

Table 4.3: Euclidean Distance error in the 3D relative positions estimated from 100 images (lowest is best).

4.2.6 Prediction quality with variation in the input resolution

The experiments above only approximate how the quantization affects performance of a model given an image of a particular input resolution. Additionally, it is important to evaluate how the quality of the 3D relative position estimation varies as resolution of the input image is reduced. The models must be evaluated for the 3D relative position computed from their predictions against the ground-truth information to observe the accuracy as a function of resolution. To perform such an evaluation, the test images from the real image test dataset described in section 3.3 are used as these images have ground-truth annotations for center pixel of the neighbors and the distance to them. For resolutions other than 320×320, the test images are resized using the nearest neighbor interpolation to preserve pixel information. The annotations for the center pixel are estimated based on the ratios of the new height and width to the original dimensions of 320×320. The annotations for the distance is preserved. The 3D relative position is calculated using eq. (2.3) (and camera intrinsic parameters obtained from calibration process described in section 3.8).

$$Error = \sqrt{(x_{FP} - x_{GT})^2 + (y_{FP} - y_{GT})^2 + (z_{FP} - z_{GT})^2}, \tag{4.3}$$

where (x_{FP}, y_{FP}, z_{FP}) and (x_{GT}, y_{GT}, z_{GT}) refer to the 3D position of the detected neighbor computed from the FP model predictions and from the annotations in the real image dataset respectively. The error in the 3D relative position estimation computed as per eq. (4.3) is shown below :

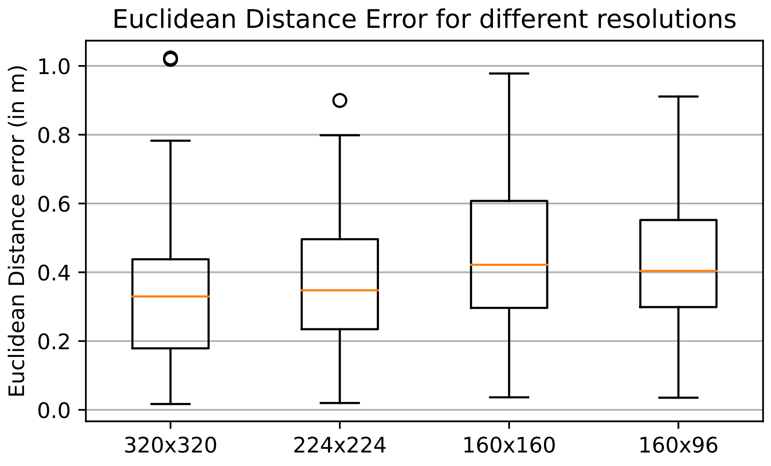


Figure 4.16: Euclidean Distance error in the 3D relative positions estimated from 97 images.

performance of the 320 × 320 was . The mean and standard deviations of the error for the 97 real images is reported in table 4.4:

	Euclidean Error	
Input Resolution	μ	σ
320 × 320	**0.331**	0.198
224 × 224	0.367	0.176
160 × 160	0.451	0.197
160 × 96	0.435	0.206

Table 4.4: Euclidean Distance error in the 3D relative positions estimated from 97 real images (lowest is best).

4.3 Discussion

In the following section, the results from the above experiments are discussed. The choice of the model architecture and the hardware configurations both influence the performance of real-time relative localization. A trade-off has to be made between the prediction quality and the inference rate that can be achieved based on the application. The following table summarizes the model performance as the resolution is changed, and the best performance for each criteria is highlighted:

Table 4.5: Model performance with different input resolutions.

	320 × 320	224 × 224	160 × 160	160 × 96
Inference speed (frames/s)	9.30	11.40	14.99	**16.10**
Mean Euclidean Distance Error (against ground-truth)	**0.331**	0.367	0.451	0.435
Mean Euclidean Distance Error (due to quantization)	0.047	**0.046**	0.08	0.103

The performance of these models against ground-truth form a baseline for evaluating their prediction quality. It was observed that reducing the input resolutions from 320×320 to 224×224 leads to approximately 11% increase in the errors of 3D position estimation. However both models have almost equal drops in prediction quality upon quantization. Using the model with the lower resolution enables performing inferences at approximately 22% higher speed than 320×320. Additionally, the on-chip memory requirements of the 224×224 model is around 0.5x the memory requirements for the 320×320 model (table 4.1). Both the 160×160 and the 160×96 models have a significant drop in prediction quality when compared to the ground-truth (approximately 36% and 31% higher errors). Their quantization errors are also significantly higher as compared to the two larger resolutions. Even though these models achieve around $60 - 70\%$ higher inference speed and have significantly lower memory requirements, they may not be very well suited for applications where a reliable 3D relative position estimation is required.

In general, the choice of the input image resolutions should be based on the requirements of the application. For a task with highly accurate but sporadic relative localization, it is favorable to choose the highest input resolution. But if the objectives require higher inference speed with tolerable error in relative localization, possible in a highly dynamic environment, then using images of 224×224 is the favorable choice.

An important factor to account for, is the scalability of these models for relative localization in multirotor teams. Due to the grid-based mechanism of the neural network, the model may be capable of detecting multiple neighbors in the image, even if trained on only images with a single visible neighbor. Each grid is treated as a small independent chunk of the image and has independent confidence and distance predictions. On scaling the relative localization method to multirotor teams, there is a possibility that the prediction quality reduces. If the neighbors are located in close proximity and hence occupy the same grid cell in the neural network output, one of the multirotor may go unnoticed. The quality may reduce further if input images of lower resolutions are used.

5 Conclusion

Multirotors have demonstrated their versatile abilities in applications such as search-and-rescue, inspection, and delivery among others. Pocket-sized multirotors working together in a team increase the capabilities of a single multirotor. For autonomous cooperative tasks, these multirotors must have the ability to estimate the relative positions of their neighbors in real-time while relying only on their onboard computing units. Thus, the relative localization methods employed must adhere to the resource constraints of the small ultralow-powered processors on the multirotors. Vision-based deep neural networks have proven successful in predicting relative positions; however, they are naturally memory and computation intensive for the processors on board these small multirotors.

In this work, a vision-based deep neural network for relative localization was quantized and deployed on a light-weight quadcopter with an ultralow-power parallel computing processor and a grayscale camera. Variants of the model with different input resolutions were implemented and the effect of quantization was studied. Quantization leads to a small drop ($\approx 4 - 10\%$ errors introduced based on the variant) in the prediction quality of the models but these are tolerable trade-offs for real-time onboard capabilities. The performance of the deployed models were analyzed further for the absolute prediction quality and inference speed criteria. Experiment results show that using images of higher resolution guarantees a better prediction quality than images of lower resolutions at the cost of processing lesser number of images per second. Thus, trade-off must be considered when choosing between higher image resolution for improved accuracy and lower image resolution for faster inference speed. Other small, but important criterion like the memory required to store the parameters of the network, the number of MAC operations and the frequency configuration of the computing units also influence the choice of the model for real-time relative localization.

The models presented have the capability for multi-robot relative localization, however there may be some reduction expected in prediction quality if the neighbors are located in close proximity. Further drop in prediction quality can be expected if the input image resolutions are reduced. The inference speed, can be expected to remain approximately the same as in the findings of the thesis. This is because the computations of the neural network account for the majority of the inference time, which due to the grid-based network output remains the same for multiple neighbors. There might be a slight addition to the inference time in the post-processing of the network output but that can be compensated by parallelizing the 3D position estimation on the STM32 MCU while GAP8 computes on the inference on the next image. Further developments may include improving the deep neural network for multi-robot detections and localization for efficient swarming.

Appendix A

Stagewise quantization performance evaluation

The stage-wise performance during quantization of the models is shown. The FP refers to the *FullPrecision* model, FQ to the *FakeQuantization* model, QD to *QuantizedDeployable* model and ID to *IntegerDeployable* model. GT refers to the ground truth annotations from the dataset.

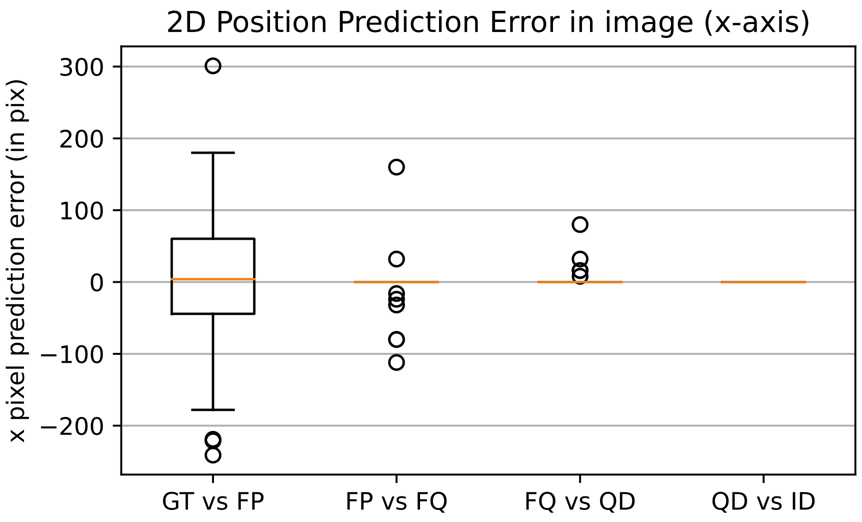


Figure A.1: 2D position (x-axis) error in image across different stages of quantization for 97 real images dataset.

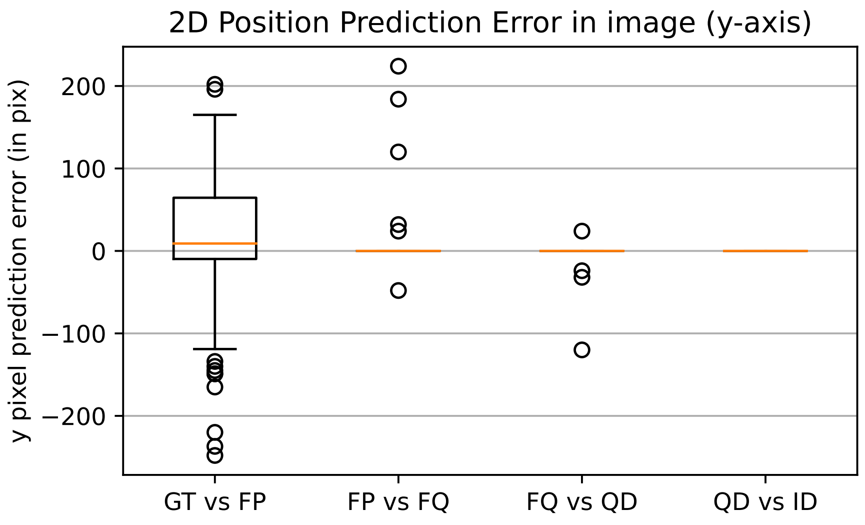


Figure A.2: 2D position (y-axis) error in image across different stages of quantization for 97 real images dataset.

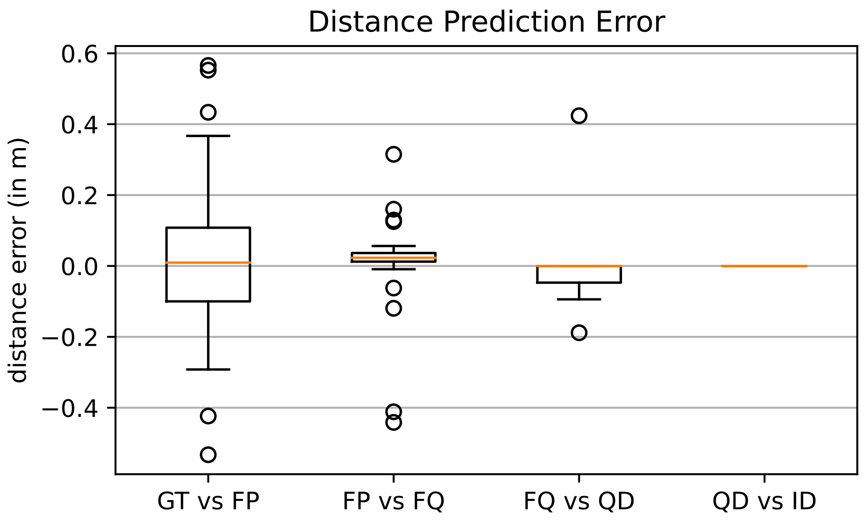


Figure A.3: Distance error across different stages of quantization for 97 real images dataset.

The distance predictions are most affected by quantization. This is because unlike a pixel coordinate which is naturally an integer, the real-valued distance metric loses precision when converted to integers.

Appendix B

Additional GAP8 performance metrics of onboard inference

Table B.1 shows the time taken by the GAP8 processor for a single frame inference. By increasing the number of active cores, faster inference times can be achieved. As expected, keeping all cores active, the inference times for larger input resolutions are higher ($\approx 15x$) than the inference times for smaller input resolutions.

Input Resolution	Cores			
	1	2	4	8
320×320	220.61	155.74	123.51	107.48
224×224	143.40	111.31	95.50	87.71
160×160	90.23	73.70	65.60	61.60
160×96	24.63	14.56	9.70	7.45

Table B.1: Single frame inference time (in ms) vs Cores for different resolutions.

Input Resolution	Cores				Speed-up vs.
	1	2	4	8	1-core
320×320	13.28	6.79	3.57	1.97	6.7x
224×224	6.58	3.37	1.78	1.01	6.5x
160×160	3.40	1.74	0.93	0.53	6.4x
160×96	2.03	1.04	0.56	0.32	6.3x

Table B.2: Application Performance [Mcycles] vs Cores for different resolutions.

The cycles-counts capture the CPU time and also DMA accesses. Hence change in the number of parameters of the network by reducing the input resolution influences the cycle-counts [6].

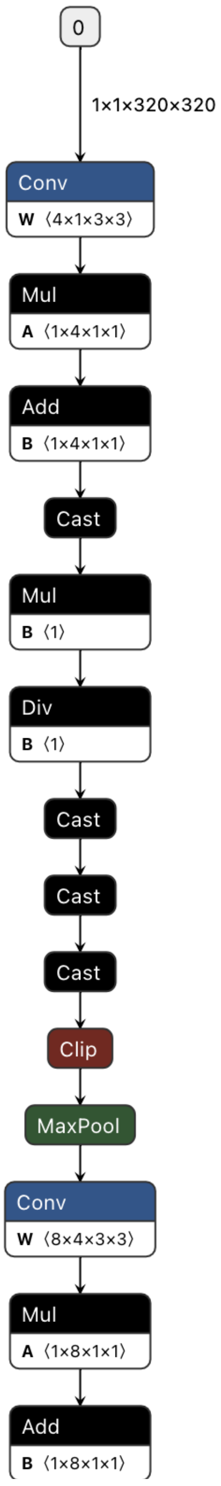
Appendix C

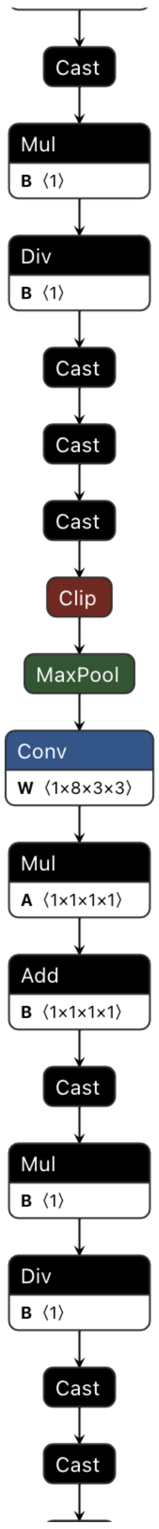
ONNX representation

C.1 Full ONNX representation of the model with input 320x320 images

This figure was visualized using Netron[1], a viewer for neural network.

[1]`https://github.com/lutzroeder/netron`


0
1×1×320×320
Conv
W ⟨4×1×3×3⟩
Mul
A ⟨1×4×1×1⟩
Add
B ⟨1×4×1×1⟩
Cast
Mul
B ⟨1⟩
Div
B ⟨1⟩
Cast
Cast
Cast
Clip
MaxPool
Conv
W ⟨8×4×3×3⟩
Mul
A ⟨1×8×1×1⟩
Add
B ⟨1×8×1×1⟩


Cast
Mul
B ⟨1⟩
Div
B ⟨1⟩
Cast
Cast
Cast
Clip
MaxPool
Conv
W ⟨1×8×3×3⟩
Mul
A ⟨1×1×1×1⟩
Add
B ⟨1×1×1×1⟩
Cast
Mul
B ⟨1⟩
Div
B ⟨1⟩
Cast
Cast

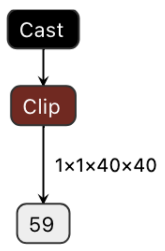


Figure C.1: Full ONNX representation of the model with input 320 × 320 images.

Bibliography

[1] Meysam Basiri et al. "Audio-based localization for swarms of micro air vehicles". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 4729–4734. DOI: 10.1109/ICRA.2014.6907551 (cit. on p. 5).

[2] Stefano Bonato et al. *Ultra-low Power Deep Learning-based Monocular Relative Localization Onboard Nano-quadrotors*. 2023. arXiv: 2303.01940 [cs.RO].

[3] Alessio Burrello et al. "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs". In: *IEEE Transactions on Computers* 70.8 (2021), pp. 1253–1268. DOI: 10.1109/TC.2021.3066883 (cit. on pp. 14, 20, 21, 25, 38, 40).

[4] Rene Y Choi et al. "Introduction to Machine Learning, Neural Networks, and Deep Learning". In: *Translational Vision Science & Technology* 9.2 (Feb. 2020), pp. 14–14. DOI: 10.1167/tvst.9.2.14 (cit. on p. 8).

[5] Francesco Conti. *Technical Report: NEMO DNN Quantization for Deployment Model*. 2020. arXiv: 2004.05930 [cs.LG] (cit. on pp. 13, 15, 16, 25, 33).

[6] Eric Flamand et al. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445101 (cit. on pp. 14, 66).

[7] Dario Floreano and Robert J. Wood. "Science, technology and the future of small autonomous drones". In: *Nature* (2015). DOI: https://doi.org/10.1038/nature14542 (cit. on p. 5).

[8] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV] (cit. on p. 13).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on p. 10).

[10] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: `1704.04861`. URL: `http://arxiv.org/abs/1704.04861` (cit. on p. 18).

[11] Andrey Kuzmin et al. *Pruning vs Quantization: Which is Better?* 2023. arXiv: `2307.02973 [cs.LG]` (cit. on p. 12).

[12] Shushuai Li, Christophe De Wagter, and Guido C. H. E. de Croon. *Self-supervised Monocular Multi-robot Relative Localization with Efficient Deep Neural Networks*. 2021. arXiv: `2105.12797 [cs.RO]` (cit. on pp. 6, 7, 24, 26, 29, 30).

[13] Shushuai Li et al. *An autonomous swarm of micro flying robots with range-based relative localization*. 2021. arXiv: `2003.05853 [cs.RO]` (cit. on pp. 2, 5).

[14] Aaron Lopez Luna, J. Martinez-Carranza, and Israel Cruz Vega. "Towards Aerial Interaction of MAVs in GPS-Denied Environments". In: *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*. 2019, pp. 113–121. DOI: `10.1109/REDUAS47371.2019.8999686` (cit. on p. 5).

[15] Akmaral Moldagalieva and Wolfgang Hönig. *Virtual Omnidirectional Perception for Downwash Prediction within a Team of Nano Multirotors Flying in Close Proximity*. 2023. arXiv: `2303.03898 [cs.RO]` (cit. on pp. 6, 7, 24, 28, 29, 41, 44).

[16] Vlad Niculescu et al. "Improving Autonomous Nano-Drones Performance via Automated End-to-End Optimization and Deployment of DNNs". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (2021), pp. 548–562. DOI: `10.1109/JETCAS.2021.3126259`.

[17] Daniele Palossi et al. *Fully Onboard AI-powered Human-Drone Pose Estimation on Ultra-low Power Autonomous Flying Nano-UAVs*. 2021. arXiv: `2103.10873 [cs.RO]`.

[18] Aurello Patrik et al. "GNSS-based navigation systems of autonomous drone for delivering items". In: *Journal of Big Data* 6.1 (June 2019), p. 53 (cit. on p. 2).

[19] Eric Price et al. *Deep Neural Network-based Cooperative Visual Tracking through Multiple Micro Aerial Vehicles*. 2018. arXiv: `1802.01346 [cs.RO]` (cit. on p. 6).

[20] Antonio Pullini et al. "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing". In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: `10.1109/JSSC.2019.2912307` (cit. on p. 13).

[21] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: `1804.02767 [cs.CV]` (cit. on p. 26).

[22] James Roberts et al. "3-D Relative Positioning Sensor for Indoor Collective Flying Robots". In: *Autonomous Robots* 33 (Aug. 2012). DOI: `10.1007/s10514-012-9277-0` (cit. on p. 4).

[23] Steven Roelofsen, Denis Gillet, and A. Martinoli. "Reciprocal collision avoidance for quadrotors using on-board visual detection". In: Sept. 2015, pp. 4810–4817. DOI: `10.1109/IROS.2015.7354053` (cit. on p. 6).

[24] Leticia Oyuki Rojas Pérez and Jose Martinez-Carranza. "Flight Coordination of MAVs in GPS-denied Environments using a Metric Visual SLAM". In: Oct. 2019 (cit. on p. 5).

[25] Manuele Rusci, Alessandro Capotondi, and Luca Benini. *Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers*. 2019. arXiv: `1905.13082 [cs.LG]` (cit. on p. 33).

[26] Fabian Schilling, Fabrizio Schiano, and Dario Floreano. "Vision-Based Drone Flocking in Outdoor Environments". In: *IEEE Robotics and Automation Letters* PP (Feb. 2021), pp. 1–1. DOI: `10.1109/LRA.2021.3062298`.

[27] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch*. 2020. URL: `https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf` (cit. on p. 10).

[28] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: `10.1109/JPROC.2017.2761740` (cit. on pp. 9–11).

[29] Matouš Vrba and Martin Saska. "Marker-Less Micro Aerial Vehicle Detection and Localization Using Convolutional Neural Networks". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 2459–2466. DOI: `10.1109/LRA.2020.2972819` (cit. on p. 6).

[30] Alexander Woods and Hung La. "Dynamic Target Tracking and Obstacle Avoidance using a Drone". In: Dec. 2015. ISBN: 978-3-319-27856-8. DOI: `10.1007/978-3-319-27857-5_76` (cit. on p. 4).

[31] Hao Wu et al. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. arXiv: `2004.09602 [cs.LG]`.