Modular Neural Network Navigation for Autonomous Nano Drone Racing

F. Magri

TU Delft

Delft University of Technology

Modular Neural Network Navigation for Autonomous Nano Drone Racing

by

F. Magri

Student Name	Student Number
Federico Magri	4663438

Supervisors: Christophe De Wagter, Guido C.H.E. de Croon, Robin Ferede, Stavrow Bahnam
Thesis Duration: February, 2023 - December, 2023
Faculty: Faculty of Aerospace Engineering, Delft University of Technology

Cover: FAI World Drone Racing Championship (Modified)

TU Delft

Preface

I am immensely grateful for the support and guidance I have received throughout the course of this work, and it is with great appreciation that I acknowledge those who have played a pivotal role in my journey.

To my supervisors, Christophe, Guido, Stavrow and Robin, thank you for your guidance and patience. From the meetings, always overtime, to invaluable advice.

To my parents and brother, your support has been my constant source of strength. You have always believed in me, even when the path seemed uncertain, and for that, I am grateful.

And to my friends, thank you for standing by me, for all the moments of laughter and understanding that have lightened the load of this journey.

This accomplishment is not mine alone, but a testament to the collective support of each one of you. Thank you.

F. Magri
Delft, December 2023

Abstract

In this study, we present a first step towards a cutting-edge software framework that will enable autonomous racing capabilities for nano drones. This is done through the integration of neural networks tailored for real-time operation on resource-constrained devices. A lightweight Convolutional Neural Network, with the Gatenet architecture, is adjusted for reduced computational demand and is successfully deployed on a GAP8 processor at a rate of 16Hz. This network provides gates' size and location data for the subsequent positioning algorithm. A second neural network, trained through reinforcement learning, governs the drone's guidance and control systems, demonstrating a remarkable rate of 167Hz on an STM32F405 processor. The attitude rates and thrust outputted by this network are then fed to an attitude rate PID controller.

The research shows that state-of-the-art neural networks for drone racing can be deployed on nano drones, despite their limited processing power. Nonetheless, the study demonstrated specific limitations, such as the perception network's sensitivity to white pixels in the image reducing its effectiveness when light sources are present in the scene. These findings underscore the importance of dataset composition and the need for diverse training scenarios to enhance the neural network's generalizability and performance in real-world applications.

Contents

Nomenclature

Abbreviations

CNN	Convolutional Neural Network
EKF	Extended Kalman Filter
GPS	Global Positioning System
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
INDI	Incremental Nonlinear Dynamic Inversion
MPC	Model Predictive Control
PID	Proportional Integral Derivative
RANSAC	Random Sample Consensus
VIO	Visual-Inertial Odometry

Symbols

$\boldsymbol{\omega}$	Angular velocity
λ	Attitudes
$\mathbf{\Omega}$	Attitude Rates
$\mathbf{C}$	Center of Camera
$\mathbf{d}$	Distance from camera to point
$\mathbf{E}$	Errors
$\mathbf{e}$	Basis vector of the world coordinate systems
$\mathbf{P}$	Point in image
$\mathbf{p}$	Position
$\mathbf{R}$	Rotation Matrix
$\mathbf{t}$	Translation Vector
$\mathbf{v}$	Velocities
$\mathcal{L}$	Loss
μ	Loss Weight
ϕ	Roll
ψ	Yaw
σ	Threshold
τ	Time Constant

θ	Pitch
$\tilde{\phi}$	Variance of heading from gate
$\tilde{\psi}$	Variance of distance from gate
$\tilde{\theta}$	Relative heading from gate
$\tilde{r}$	Mean distance from gate
ζ	Camera Angle
a	Acceleration
b, n	Biases
c	Confidence for gate
d	Distance
F	Sate Jacobian
f	Focal Length
H	Measurement Jacobian
K	Kalman Gain
M	Scale
N	Number of Pixels
P	Covariance Matrix
p	Roll Rate
Q	Process Noise Covariance
q	Pitch Rate
q_q	Quantized value
r	Yaw Rate
$r(k)$	Reward
r_q	Real value
S	Residual Covariance
s	Size
s_q	Quantization Parameter
T	Thrust
u	Center Pixel Horizontal
v	Center Pixel Vertical
v	Velocity
x^b	Binarised variable
Z	Zero-point
x_k	State Vector
z_k	Measurement Vector

1 Introduction

In the intersection of deep learning and autonomous systems, the world of autonomous drone racing has emerged as a testing bed for cutting-edge technology. The last few years have witnessed tremendous advancements in deep learning, driven by new reinforcement learning strategies and sophisticated neural network architectures. These technological leaps have consistently outperformed human expertise in simulations and controlled gaming contexts, such as Go[9], Starcraft[11], and chess[9]. Yet, the application of such artificial intelligence has been predominantly within simulations, manily due to the less predictable variables of real-world settings.

The world of drone racing where trained pilots maneuver quadcopters at speeds exceeding 100 km/h, has recently been recognized as a prime environment for testing and enhancing AI systems. The Alphapilot autonomous racing competition in 2019 was a showcase of the cutting-edge research being conducted in this domain. However, human pilots were still significantly faster, with drones operated by AI lagging by a factor of at least two[2]. A paradigm shift occurred with the introduction of the Swift system, which, powered by a Nvidia Jetson TX2, successfully outperformed multiple professional human pilots for the first time in history[6]. This field's relevance extends beyond sport; the technologies developed for drone racing have implications for real-world applications like search and rescue operations and autonomous inspections in disaster-struck regions where indoor navigation without GPS is vital[5].

Despite the progress, a significant gap remains in deploying neural networks, the state-of-the-art for control and perception[5], on edge devices like nano drones. These devices face tight constraints in computational resources and latency, that are amplified when drones must navigate at high speeds. Addressing these challenges is not trivial, as traditional neural networks are not feasible on such constrained devices, as they usually run on powerful GPUs[2].

This thesis delves into a system designed to race a 40g nano drone, capable of running AI algorithms on processors with limited computational power. It describes the development and deployment of two lightweight deep neural networks: one for perception and gate detection and another for guidance and control, both optimized through parameter tuning and quantization. Additionally, this report provides a comprehensive review of state-of-the-art strategies employed at each stage of the autonomous drone racing software stack. From perception algorithms converting images to position and state[7, 8, 10], to planning and control methods[1], and methods to deploy neural networks for reduced latency[4].

Embarking from this foundation, the main research question this work seeks to answer is:

How can state-of-the-art perception and control algorithms for autonomous drone racing be simplified to fit nano-drones, specifically a crazyflie equipped with AI-Deck?

In addressing the central research question, this study has yielded three significant contributions that address the existing knowledge gap in the field of autonomous nano drone racing. The first is the successful development of a neural network capable of accurately locating racing gates at very low image resolutions. The second contribution is the effective deployment of this network on the AI-Deck, achieving inference speeds sufficient for real-time drone flight. Lastly, the study has pioneered the implementation of a lightweight guidance and control neural network, trained via reinforcement learning, on the flight controller of a Crazyflie drone, allowing autonomous flight to happen.

This thesis document is divided in three parts: Part.I contains the scientific paper, Part.II contains additional results which dig into more details about certain aspects of the project, and finally Part.III, the literature review. All the code for this project can be found on github[1]

[1]https://github.com/fed12345/nano-drone-racing

Part I

Scientific Paper

Modular Neural Network Navigation for Autonomous Nano Drone Racing

Federico Magri, Robin Ferede, Stavrow Bahnam, Christophe De Wagter, Guido C.H.E De Croon

***Abstract*—In this study, we present a first step towards a cutting-edge software framework that will enable autonomous racing capabilities for nano drones. Through the integration of neural networks tailored for real-time operation on resource-constrained devices. A lightweight Convolutional Neural Network, with the Gatenet architecture, is adjusted for reduced computational demand and is successfully deployed on a GAP8 processor at a rate of 16Hz. This network provides gates' size and location data for the subsequent positioning algorithm. A second neural network, trained through reinforcement learning, governs the drone's guidance and control systems, demonstrating a remarkable rate of 167Hz on an STM32F405 processor. The attitude rates and thrust outputted by this network are then fed to an attitude rate PID controller.**

The research shows that state-of-the-art neural networks for drone racing can be deployed on nano drones, despite their limited processing power. Nonetheless, the study demonstrated specific limitations, such as the perception network's sensitivity to white pixels in the image reducing its effectiveness when light sources are present in the scene. These findings underscore the importance of dataset composition and the need for diverse training scenarios to enhance the neural network's generalizability and performance in real-world applications.

***Index Terms*—Reinforcement learning, Convolutional neural network, Quantization, Drone Racing**

I. Introduction

DEEP Learning has seen incredible progress over the last years, with notable advancements propelled by the integration of reinforcement learning and deep neural network architectures. These methods have proven their worth by surpassing human experts in several competitive areas, including games like Go [1], Starcraft [2], and chess [1]. Despite these successes, the application of Artificial Intelligence(AI) has remained predominantly within the bounds of simulations and controlled gaming contexts, lacking the unpredictable variables present in real-world settings.

The world of drone racing, where trained pilots maneuver quadcopters at speeds exceeding 100 km/h, has recently been recognized as a prime environment for testing and enhancing AI systems. The Alphapilot autonomous racing competition in 2019 was a showcase of the cutting-edge research being conducted in this domain. However, human pilots were still significantly faster, with drones operated by AI lagging by a factor of at least two [3]. A paradigm shift occurred with the introduction of the Swift system, which, powered by a Nvidia Jetson TX2, successfully outperformed multiple professional human pilots for the first time in history [4].

With the rise of edge artificial intelligence, complex algorithms can be run on smaller drones [5]. Nano drones, with diameters under $10cm$, are an agile and versatile robotic platform employed in cases where size is of the essence. Spanning from aerial inspection tasks in narrow places [6] to human-robot interaction tasks [7]. Given the limited flight time from the battery size, the faster these drones can fly the more efficient the application is.

This article describes a system that can be used to autonomously race on nano drones, using artificial intelligence. Two lightweight deep neural networks are introduced in this article. The perception neural network detects the gates and is trained with supervised learning. The output of this network is the center and size of the gate similar to an approach taken by Pham et al. [8]. These outputs are then used to estimate the relative position. The second neural network is for guidance and control, it commands the attitude rates and thrust of the drone. This network is trained via reinforcement learning. Both networks are accelerated via tuning of the number of parameters and quantization.

The three major contributions of this article are the successful development of a neural network capable of accurately locating racing gates at very low image resolutions. The deployment of this network on the AI-Deck, achieving inference speeds sufficient for a nano drone to fly. Finally, the implementation of a lightweight guidance and control neural network, trained via reinforcement learning, on the flight controller of a Crazyflie drone.

II. Related work

Advanced algorithms such as Visual Inertial Odometry [9] or feature-based Simultaneous Localization And Mapping (SLAM) [10] are unviable due to their intensive computational demands. Similarly, the application of control strategies, like Model Predictive Control [11], are discarded because of their computational intensity.

Consequently, we move towards employing Neural Networks, prioritizing lightweight ones, to avoid overburdening the drone's processing capabilities. DroNet proposed an end-to-end navigation solution via a single neural network [12]. However, our approach leans towards a modular system as it tends to be faster and easier to debug. A faster approach is because the perception network and control network can operate at independent speeds, which is particularly advantageous for certain needs, such as the control network running at a much faster rate. Modularity also offers the benefit of providing insights into the actions of each network, rather than treating the system as a black box.

The state-of-the-art system for drone racing is Swift [4] and also follows a modular neural network approach. On the

perception side, this system runs a gate detection Convolutional Neural Network(CNN), that infers the corners of gates and the part affinity fields [13]. Following this, the gates are identified and the position of the drone on the race track is determined. This network has a U-net structure not compatible with nano drones due to its computational load. For this reason, lightweight and efficient networks, like Gatenet [8] and Dronet [12] are a better option for this project.

On the control side, Swift employs a two-layer network that maps the output of the Kalman filter to the control commands for the drone. The policy is trained using reinforcement learning in simulation [4]. During training, the policy maximizes a reward that combines progress towards the next racing gate with a perception objective that rewards keeping the next gate in the field of view of the camera. This approach is also used in the Guidance and Control Net [14], with the output of the network being the motor rpm commands directly instead of the attitude rates and thrust and no perception penalty. Deploying reinforcement learning policies on nano drones is a challenge due to their unpredictability, as there will be a large reality gap between the simulator and the real world. This reality gap is attributed to many factors namely, the sensitivity to disturbances due to the small size of nano drones, the dependency on the battery state [5], and the fact that the Crazyflie motors are brushed.

III. Hardware

To devise an optimal racing strategy for nano drones, one must first consider the inherent constraints of these small devices. We employ the CrazyFlie 2.1 as our primary model due to its widespread availability and its open-source development platform. This drone is powered by an STM32F405 processor (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash) and utilizes the nRF51822 for radio communications and power management (Cortex-M0, 32MHz, 16kb SRAM, 128kb flash). Additionally, it has 3-axis accelerometers/gyroscopes and a high-resolution pressure sensor [15]. Moreover, the CrazyFlie 2.1 is equipped with an AI-Deck that integrates a Hi-Max Gray-scale camera, with a low-power GAP8 processor.

IV. Implementation

The software stack is shown in figure 1. One can notice that the perception task is handled by the AI-Deck, where a [122x162] pixel image is captured by the Hi-Max camera at a rate of 30Hz. This image is then fed to a network that detects gates and outputs the center coordinates of the gate and the apparent size, in pixels. Following this, a relative position algorithm estimates from these parameters, the relative position from the gate to the drone. In this research, the state estimation is done with a motion capture system and an IMU. Alternatively, one could use the relative gate position and IMU but this is outside the scope of the project.

Once the states have been estimated, a Guidance and Control network [14] outputs the desired thrust and attitude rates, and will fly the drone through the gate. At the lowest level, the rates are fed to a PID controller, developed by Bitcraze, which outputs the motors' rpms.

A. Perception

1) Architecture: In order to detect a gate, a CNN-based approach is taken. Two state-of-the-art neural networks are compared in this work. The first one, GateNet [8], is a CNN with 6 convolutional layers followed by a max pooling, and at the end a fully-connected layer, illustrated in figure 2. The idea of this network would be for the convolutional layers to detect features and the fully connected layer to map these features to the pixel coordinates of the center, the width, and the height of the gate. The other network is Dronet [12], which is a variant of the ResNet-8 neural network with three residual blocks. This architecture is efficient and simplifies back-propagation. In addition, these residual blocks help combat the vanishing gradient problem [12]. The outputs of these networks are the u and v coordinates of the center of the gate, the height and width of the gate(in pixels), and a confidence value that determines if a gate is in the image or not.

2) Training: Running a neural network on the AI-Deck requires it to be lightweight, especially if it runs for drone racing purposes the latency needs to be low, leading to the need for a low-resolution input image. For this project, the networks are trained on two datasets. The first dataset consists of 5000 simulated images with the AIRR Gate [3] to compare the two network architectures and evaluate if detecting gates is possible at low resolutions[1]. The other dataset is a set of images taken with the Hi-Max camera of the gate that needs to be detected, see figure 3. This is a collection of 2000 images, of which 400 have been labeled by hand. The rest are labeled with a computer vision algorithm that detects white pixels in the image. To be automatically labelled the gate has to be on a dark background.

3) Loss Function: The network is subject to minimizing the loss function in equation 1.

$$\begin{aligned}\mathcal{L} =& \mu_{\text{center}}\hat{c}\left((u-\hat{u})^2+(v-\hat{v})^2\right)\\ &+\mu_{\text{size}}\hat{c}\left(\left(s_u-\hat{s}_u\right)^2+\left(s_v-\hat{s}_v\right)^2\right)+\mu_{\text{conf}}(c-\hat{c})^2,\end{aligned} \tag{1}$$

where $\mu_{\text{center}}=2$, $\mu_{\text{size}}=2$ and $\mu_{\text{conf}}=1$, are the weights of the center coordinate, size and confidence parts of the loss function, these are non-trainable. (u,v) and $(\hat{u},\hat{v})$ are the coordinates to the center of the gate, predicted and ground-truth respectively. s_u, s_v are the width and height of the gate in pixels.

4) Deployment: The GapFlow process, pipeline by Greenwaves technologies [16], is used to deploy the CNN on the GAP8 processor. Before the network can go through this pipeline it needs to be quantized. For this, the training-aware quantization of TensorFlow is employed. Network sparsification, to reduce the number of connections in a network, was also attempted but did not improve the performance of the network(see Additional Results). This model is then fed to the GAP NNTool, which maps the network nodes to the GAP8 computational nodes. Next, the GAP Autotiler optimizes the data movement across the memory hierarchy based on the

[1]https://github.com/open-airlab/pencilnet

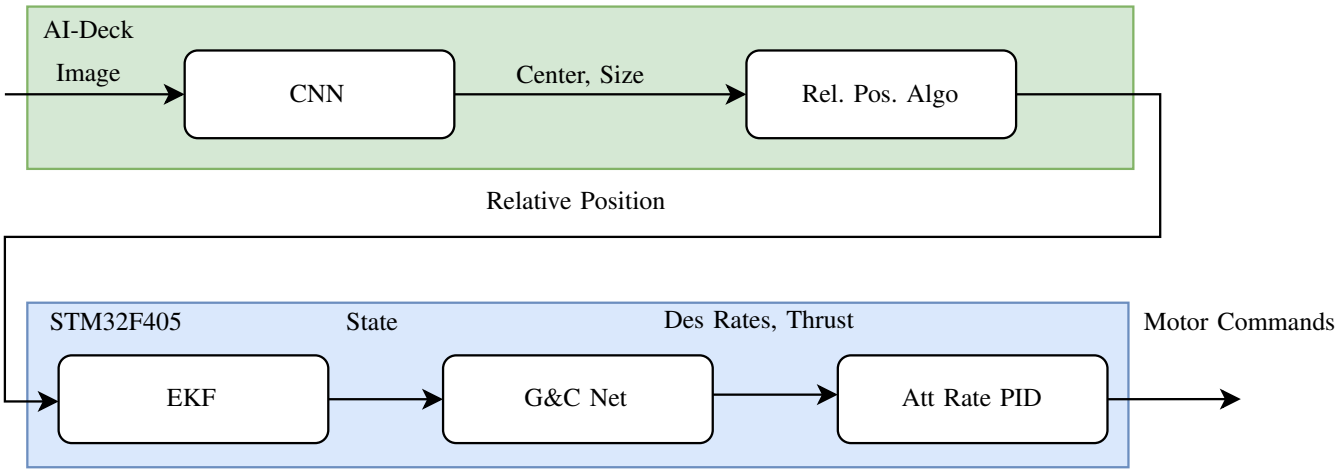


Fig. 1: Software stack for racing on the Crazyflie

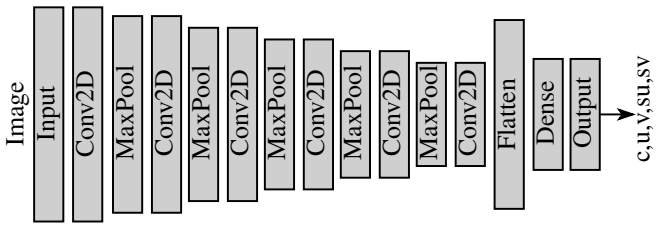


Fig. 2: Network architecture of Gatenet

Fig. 3: On the left, the test set of the simulation dataset, in green is the label and in red are results of Gatenet. On the right, two images captured by the AI-Deck camera. Again in red the output of the network, no labels are present for these images.

network and memory constraints. Finally, this generated code can be dockerized and flashed to the AI-Deck with the help of the crazyradio.

B. Relative Position

The relative position from the gate to the drone is estimated using the center coordinates and size of the gate, which are computed by the gate detection network. The first step is to calculate the angle, ζ, between the gate center and the center of the camera, shown in equation 2. Where u and v are the center coordinates of the gate and camera, in pixels. While, f is the focal length.

$$\zeta = \frac{v_{\text{gate}} - v_{\text{cam}}}{f_y}, \zeta_y = \frac{u_{\text{gate}} - u_{\text{cam}}}{f_x} \tag{2}$$

Following this, the distance to the camera is calculated according to equation 3, where s_{real} is the size of the gate, assumed to be known, in meters and s_{net} is the size of the gate in pixels, which is the output of the gate detection network. The distance can be calculated using s_u or s_v values according to which measurement is larger. The camera focal length and center are determined experimentally.

$$d = s_{\text{real}_x} \times \left(\frac{f_x}{s_{\text{net}_u}} \right) \tag{3}$$

Finally, x, y, and z relative position from the center of the gate to the drone are extracted using equations 4, 5 and 6. The coordinate system is z up with the origin located at the center of the gate, the x-axis is aligned with the direction the gate needs to be crossed. Thus, the drone is always behind the gate. The major drawback of this approach is that it assumes the relative yaw between the gate and the drone to be zero.

$$x = -d \tag{4}$$
$$y = -\zeta_x \times d \tag{5}$$
$$z = -\zeta_y \times d \tag{6}$$

C. Control

Once the relative position has been determined, a feed-forward network estimates the desired attitude rates and thrust, then a PID controller yields the motor rpms'. To achieve this we train the model in simulation using reinforcement learning.

The pipeline begins with the drone being modeled to have a good simulator, next the architecture of the network was decided. Following this, the network was trained in simulation leveraging the Markov Decision Process(MDP) framework. Finally, the network is deployed on the flight controller. To fly the control network an imaginary race is set up, with four gates in a 4×3m rectangle. The goal is to fly this path as quickly as possible.

1) Quadcopter Model: The controller's response to commands are assumed to be a first-order delay system. The model encompasses kinematic equations and a linear drag model, the states and inputs are:

$$\mathbf{x} = [\mathbf{p}, \mathbf{v}, \boldsymbol{\lambda}, \boldsymbol{\Omega}, T]^T, \quad \mathbf{u} = [\boldsymbol{\Omega}_{\text{cmd}}, T_{\text{cmd}}]^T$$

Where $\mathbf{p}$ are the positions of the drone, $\mathbf{v}$ are the velocities, λ are the Euler angles, $\boldsymbol{\Omega}$ are the angular rates, and T is the thrust. The equations governing motion are:

$$\dot{\mathbf{p}} = \mathbf{v}, \tag{7}$$
$$\dot{\mathbf{v}} = g\mathbf{e}_3 + R(\boldsymbol{\lambda})\mathbf{F}, \tag{8}$$
$$\dot{\boldsymbol{\lambda}} = Q(\boldsymbol{\lambda})\boldsymbol{\Omega}, \tag{9}$$
$$\dot{\boldsymbol{\Omega}} = (\boldsymbol{\Omega}_{\text{cmd}} - \boldsymbol{\Omega})/\tau_{\Omega}, \tag{10}$$
$$\dot{T} = (T_{\text{cmd}} - T)/\tau_T, \tag{11}$$

From flight data we computed τ_p and τ_q to be 0.0382. τ_r to be 1.1113, and τ_T to be 0.0565,. Moreover, the specific force $\mathbf{F}$ is formulated as $\mathbf{F} = [-d_x v_x^B, -d_y v_y^B, -T]^T$. From empirical data, we derived d_x to be 0.34 and d_y to be 0.43.

To optimize performance, we set defined boundaries for thrust and attitude rates, tailored to the most stable and rapid flight achievable in our system:

$$p_{\text{cmd}}, q_{\text{cmd}} \in [-0.2, 0.2], \quad T_{\text{cmd}} \in [7.8, 10.6], , r_{\text{cmd}} \in [-0.6, 0.6]$$

2) Architecture: The policy network processes 13 inputs, which include states of the drone relative to the gate, denoted by superscripts g, attitude rates, and thrust:

$$\mathbf{x}_{\text{in}} = [\mathbf{p}^{g_i}, \mathbf{v}^{g_i}, \boldsymbol{\lambda}^{g_i}, \boldsymbol{\Omega}, T]^T$$

The network outputs four control signals: thrust and body rate commands. Distributed from -1 to 1 to simplify the network learning. The architecture consists of three general matrix multiply(GeMM) layers, with a ReLu activation function, with the same amount of neurons, see in figure 4.

3) Training: In order to train the network using reinforcement learning, we transform our dynamic model into a discrete-time MDP. This framework defines the set of states the environment can be in, the set of actions the agent(drone) can take, the reward for every given action, and the probabilities of moving from one state to the next after an action. Initial states are uniformly sampled from specified intervals, as described in equation 12. The agent starts at one of these states and employs the network to choose an action with a certain probability and receiving a certain reward, r_t.

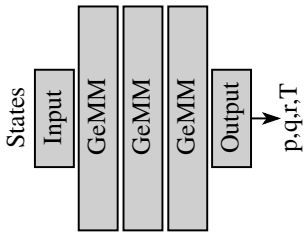


Fig. 4: Network architecture of the Guidance and Control network

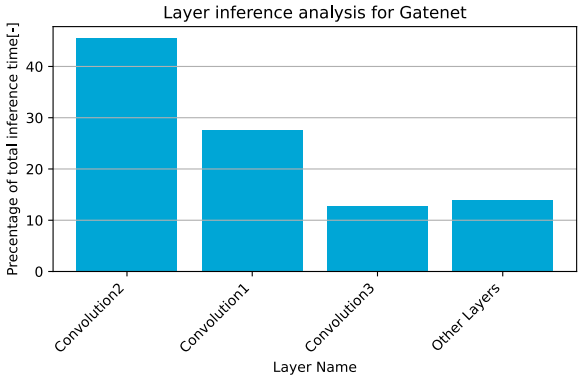


Fig. 5: Inference analysis of the layers of GateNet as a percentage of the total inference time. Inference is run on the simulation dataset

The objective is to change the parameters (w) of a neural network policy to maximize expected return $\max_w \mathbf{E}_{\pi_w}[\sum_{t=0}^{\infty} \gamma^t r_t])$.

$$\begin{array}{lll} x \in [-\frac{1}{2}, \frac{1}{2}] + x_s & y \in [-\frac{1}{2}, \frac{1}{2}] + y_s & z \in [-\frac{1}{2}, \frac{1}{2}] + z_s \\ v_x \in [-\frac{1}{2}, \frac{1}{2}] & v_y \in [-\frac{1}{2}, \frac{1}{2}] & v_z \in [-\frac{1}{2}, \frac{1}{2}] \\ \phi \in [-\frac{2\pi}{9}, \frac{2\pi}{9}] & \theta \in [-\frac{2\pi}{9}, \frac{2\pi}{9}] & \psi \in [-\pi, \pi] \\ p \in [-1, 1] & q \in [-1, 1] & r \in [-1, 1] \\ T \in [9.8, 10.1] & & \end{array} \tag{12}$$

The drone's goal is to complete an oval path, moving through each gate as rapidly as possible. The performance is evaluated using a reward function 13, as implemented by Ferede et al. [14] with the additional yaw penalty.

$$r(k) = \begin{cases} ||\mathbf{p}_k - \mathbf{p}_{g_k}|| - ||\mathbf{p}_{k-1} - \mathbf{p}_{g_k}|| & \\ -(\psi_k - \psi_{g_k})^2 & \text{if } r > 30 \\ +10 - 10||\mathbf{p}_k - \mathbf{p}_{g_k}||, & \text{if gate passed} \\ -10, & \text{if collision} \end{cases} \tag{13}$$

In this function, $\mathbf{p}_{g_k}$ indicates the center position of the current target gate. $\mathbf{p}_k$ and $\mathbf{p}_{k-1}$ represent the drone's present and preceding positions, respectively. The yaw is also taken into account as we always want the drone to face the gate, to ensure that the camera sees the gate. This reward becomes active only once the network manages to fly through 3 gates, making the

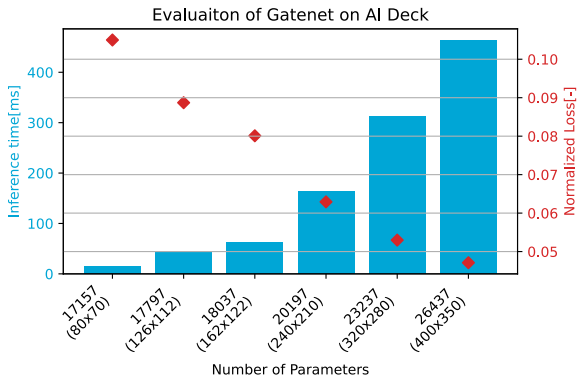


Fig. 6: Trade-off between inference time on the AI-Deck and the loss of networks with gatenet architecture with different input image sizes. Trained on the self-generated dataset.

training more effective. Successful gate passage occurs when the drone crosses the specified 1x1 meter boundary of the gate. Collisions happen either when the drone touches the ground or if it crosses outside the designated boundary of the gate.

4) Deployment: For optimal performance, the attitude rates need to be supplied at approximately $150Hz$, to receive up-to-date information but not overload the processor. To achieve this speed on an STM32F405 processor, the network needs to be lightweight. For this reason, after the training carried out with Stable Baselines Proximal Policy Optimization [17], an ONNX model is extracted and quantized with dynamic quantization, before being converted to C code[2].

V. Results

In this work, we evaluate perception and control separately. The perception networks are tailored for the AI-Deck, while the control network is deployed on the STM32F405. All flight tests were conducted in the CyberZoo, a 10x10x7 m flight arena at TU Delft's Aerospace Engineering faculty, equipped with an OptiTrack motion capture system for real-time position. The autopilot used was the Bitcraze software development kit for the Crazyflie.

A. Perception

1) Architecture Selection: Detecting gates from images of small resolution is a complex task for this reason two different neural network architectures are investigated and compared. Table I compares the Dronet [12] and Gatenet [8] architectures in terms of FPS on an Intel-i5 CPU with no multi-threading. We compare the loss according to the loss function, shown in equation 1, the error in the center coordinated ($\mathbf{E_c}$), and errors in width and height($\mathbf{E_{sx}}$ and $\mathbf{E_{sy}}$) according to equations 14 and 15 respectively.

$$E_c = |u - \hat{u}| + |v - \hat{v}| \tag{14}$$

$$E_s = |s - \hat{s}| \tag{15}$$

[2]https://github.com/kraiskil/onnx2c

The experiment found that Dronet consistently surpassed Gatenet across all error terms, while also having a lower latency. It is important to note that in the simulation dataset, all the images contain a gate therefore the confidence error is not taken into account, but can be used as a framework for further work.

Upon detailed examination, the disparity in the error metrics between the two architectures was relatively marginal. Specifically, the relative error in determining the center pixel location was 2 pixels, while the error in estimating the size of the gate was approximately 0.7 pixels. These results suggest a closer performance level between the two models than initially indicated by the loss term.

However, a critical limitation of the Dronet architecture arises from its structural complexity. Dronet's architecture cannot be quantized using either TensorFlow or the GapSDK routines. This constraint makes Dronet impractical for implementation on the AI-Deck platform. Given these technical limitations and the necessity for a balance between performance and deployability, Gatenet is further analyzed in detail in this work.

Network	FPS	Loss	$\mathbf{E_c}$[pix]	$\mathbf{E_{su}}$[pix]	$\mathbf{E_{sv}}$ [pix]
DroNet	764	83.64	8.35	1.50	0.92
GateNet	528	127.19	10.97	2.16	1.60

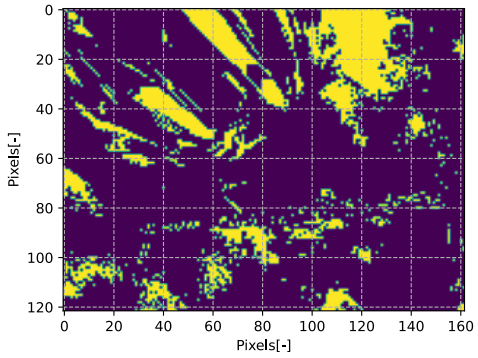
TABLE I: Comparison between DroNet and GateNet in terms of errors and inference times for images of size $[122 \times 162]$ and batch size 32. Trained and tested on the simulation dataset

2) Layer Inference Analysis: The performance of Gatenet was benchmarked on an Intel Core i5 processor, where it achieved a frame rate of 528 frames per second (FPS). Given the high-speed demands of drone racing, it is imperative to maximize the network's processing speed. The GAP8 processor, featuring a RISC-V architecture, shows architectural parallels to the x86 architecture of the Intel i5. This similarity facilitates the transferability of inference analysis between these two platforms. In Figure 5, we present a detailed breakdown of the computation time required for each layer in the Gatenet architecture, expressed as a percentage of the total inference time.

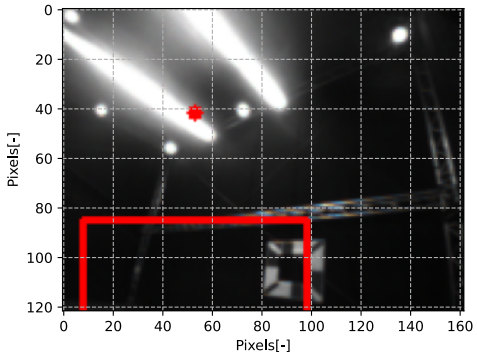
The most important observation from this analysis is that, out of the 15 layers, the first three convolutional layers account for approximately 75% of the total inference time. This significant proportion shows that these layers are the primary bottlenecks in the network's computation pipeline. Consequently, reducing the resolution of the input images leads to a substantial decrease in latency.

Another observation is that the second convolution is slower, primarily because of the significantly higher number of multiplications needed due to the increased number of channels in the input and the increased size of filters, despite the smaller height and width of the input layer.

3) Parameter Tuning: In the pursuit of optimizing the Gatenet architecture for deployment on the AI-Deck, a series of experiments were conducted to determine the most effective input image size. The primary objective was to establish a balance between inference time and accuracy, ensuring that the

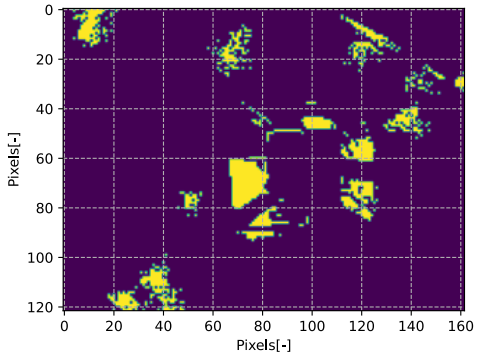


(a) Mask of the LIME algorithm, the yellow pixels are the most sensitive

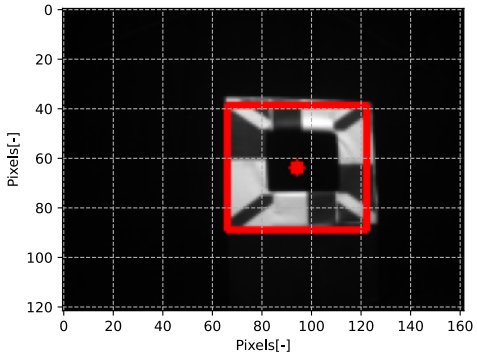


(b) Prediction of Gatenet on the image given to LIME algorithm

Fig. 7: LIME on testset image with light in the camera view



(a) Mask of the LIME algorithm, the yellow pixels are the most sensitive



(b) Prediction of Gatenet on the image given to LIME algorithm

Fig. 8: LIME on standard testset image

network maintains a high performance while operating within acceptable time constraints. The results of these experiments are shown in Figure 6. Normalized loss is the loss from equation 1 with the center and size prediction divided by the height or width of the input image.

The experimental data shows that the inference time increases with an increase in image size. For instance, with an input dimension of $[80 \times 70]$ pixels, the network ran at an inference time of 15 ms, corresponding to a frequency of approximately 66 Hz, and recorded a loss of 0.11. While a $[400 \times 350]$ image runs at $463ms[2Hz]$. This observation highlights the direct impact of image size on the computational efficiency and accuracy of the network.

After evaluating various configurations, the $[162 \times 122]$ pixel version of Gatenet was selected. This decision was informed by the requirement to keep the vision system's update rate within a minimum threshold of approximately 10 Hz.

4) LIME Algorithm: In order to better understand the results and limitations of the network. The Local Interpretable Model-agnostic Explanations(LIME) algorithm is employed [18].

LIME operates by first perturbing the input data and generating a new dataset consisting of these perturbed instances along with the corresponding predictions of the network [18]. For an individual prediction to be explained, LIME focuses on this local neighborhood generated by perturbation. It then learns a simple, interpretable model, such as a linear model, which is trained to approximate the predictions of the complex model as closely as possible within this local space.

The learned interpretable model is used to identify the importance of each feature for the prediction of the instance being explained. This is achieved by examining the weights of the linear model that lead to the decision. The features that contribute most significantly to the prediction are considered to be key explanatory factors.

The output of this algorithm is shown in figures 7 and 8. Focusing on figure 7, one can immediately notice that the network gives importance to the white sections of the gate, namely the bottom left corner and the top left corner. Apart from some noise on the side of the images, the most sensitive pixels are around the gate. Figure 7 illustrates the network's deficiencies. It depicts the white lights on the ceiling of the test environment, which correspond to white pixels that are absent

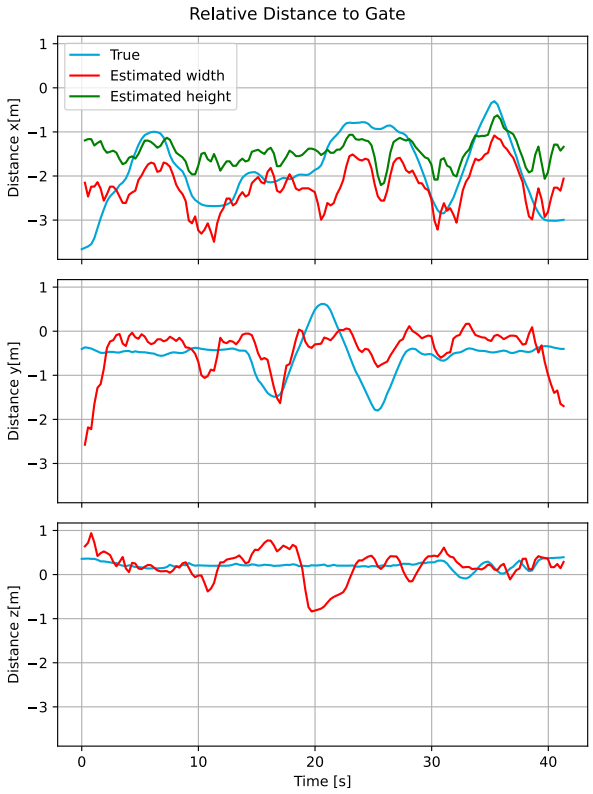


Fig. 9: Comparison between the position of the drone relative to the gate measured with a motion capture system and estimated by the CNN and relative position pipeline, estimate width is calculated with the dimension of the width and the height with the dimensions of the height. The coordinate system is z up with the origin located at the center of the gate, the x-axis is aligned with the direction the gate needs to be crossed. Thus, the drone is always behind the gate.

in the training dataset. Consequently, the network exhibits a pronounced bias towards regions with more white pixels. Additionally, the network incorrectly infers the dimensions of the gate, estimating it to be negative.

5) Relative Position: Figure 3 shows the AI-Deck running Gatenet onboard can recognize a gate and estimate its center and size in pixels. Figure 9 compares the output of the relative position onboard running onboard against the true distance from the gate to the drone over a 40-second period.

It can clearly be seen that the actual and predicted values for x are closely aligned, indicating that the network reliably predicts the gate's dimensions and any size variation. However, there is a notable discrepancy in the estimated distance when the drone is situated more than 3 meters from the gate, particularly at the start and end of the flight. This error is linked to the CNN's tendency to overreact to white pixels. Beyond the 3-meter mark, the ceiling lights in the testing environment come into the camera's view, compromising the accuracy of the predictions.

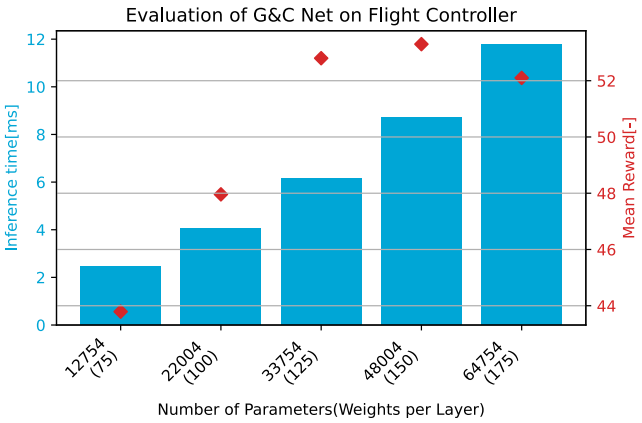


Fig. 10: Trade off between the inference time on the Crazyflie flight controller and the reward of different size networks

B. Control

1) Parameter Tuning: Once again, for the application of autonomous drone racing a faster network will result in a faster drone. For this reason, a series of experiments were conducted in which the number of neurons per layer of the network was changed, thus changing the number of parameters. These different networks were trained and the inference time on the Crazyflie flight controller was recorded.

The findings, illustrated in Figure 10, reveal a direct correlation between the network's size and its inference time, which ranged from $2.4ms$ to $11.7ms$. A particularly interesting aspect of these results is the observed plateau in the mean reward, irrespective of the network's increasing size. Despite increasing the network dimensions, the mean reward appeared to stabilize around 50. This figure suggests that, under the conditions described in the previous section, the drone averaged passing through five gates over a 12-second duration. Interestingly, while an increase in network size made the training faster, it did not increase the drone's gate navigation performance.

The Crazyflie's attitude rate PID operates at $500Hz$. For optimal performance, the attitude rates need to be supplied at approximately $150Hz$. This requirement positions the network configuration with 125 neurons as the most effective.

2) Performance Analysis: Tests were carried out on both simulated rollouts and actual flight experiments. An analysis of the real flights, as depicted in Figure 11, reveals that the overall quality of flight control is suboptimal. This deficiency is attributed to two principal factors. Firstly, there is a notable discrepancy in the drone's yaw control, as illustrated in Figure 12. The inaccuracy in modeling the drone's yaw significantly impairs the network's ability to manage this aspect of flight control, a limitation that becomes evident in the observed small looping of the drone's flight trajectory.

Secondly, the instability can be linked to the thrust model of the Crazyflie. While the thrust appears to be accurately modeled as seen in Figure 12, the underlying issue arises from how the network directly manipulates the thrust. Specifically, the network's output in Gs is transformed into a thrust command

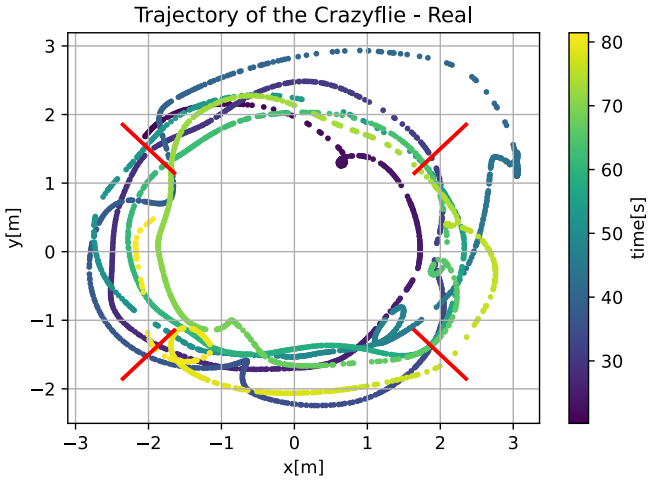


Fig. 11: Trajectory for the real flight of the Crazyflie.

via multiplication with a constant. This constant, although empirically derived, shows a high degree of sensitivity to the drone's battery state and voltage. Additionally, our model does not account for the reduced efficiency of the propellers at higher velocities, a factor that further compromises the drone's flight stability and control precision. This highlights the complex interplay between modeling accuracy and real-world control dynamics, proving the need for more sophisticated models.

Analysing the simulated flight, as depicted in Figure 13. A notable observation from the simulation data is the smoother trajectory achieved, due to the more precise control over thrust and yaw within the simulation environment. However, despite the smoother trajectory, some inconsistencies are present, especially in the first lap. These errors are less visible when the network is trained with higher limits on the commands, the reason for this is unknown and should be addressed in further research.

Another insight emerged when examining the time taken to complete the flight paths: the simulated flight accomplished five laps in 18 seconds, in contrast to the 80 seconds required for the real flight. This significant discrepancy underscores the reality gap. Specifically, in real-world scenarios, the flight is considerably slowed down by the need for numerous correction maneuvers.

Figure 12 shows the command and response of the Crazyflie during the flight. It is clear that the modeling of the drone is not very accurate, the thrust is acceptable, in this portion of the flight. While, the roll and pitch rate overshoot. Another important aspect is the yaw, as the model does not follow the behavior of the drone. Moreover, the sensor heavily undershoots the command especially in positive rotations. It is important to note that a bias was identified of $35deg/s$ between network command and response signifying an error in the PID controller, this was corrected in the flight tests.

VI. Conclusion

We showed that state-of-the-art neural networks for drone racing can be deployed on nano drones, despite their limited

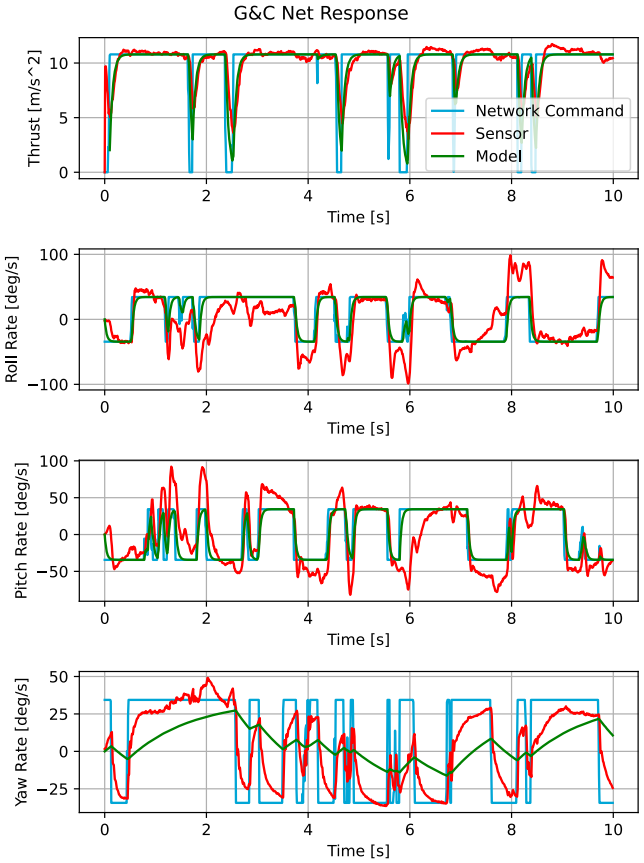


Fig. 12: Comparison between the network commands and the response of the drone over the first 10 seconds of flight.

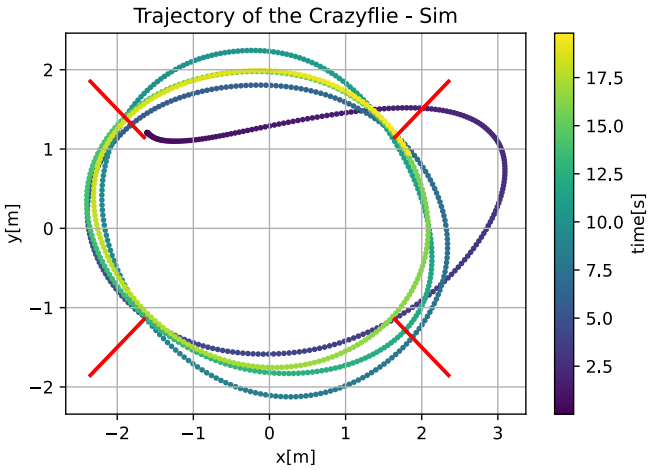


Fig. 13: Simulated trajectory of the Crazyflie.

processing power. Our approach is divided into perception and control. The former consists of a lightweight CNN with a gatenet-like architecture [8]. We have tuned and deployed this network on a GAP8 processor, achieving speeds of $16Hz$. This network outputs the size and center location of the gate which are then fed to a relative position algorithm. A second network trained with reinforcement learning is in charge of guidance and control for the drone. This network was deployed on the STM32F405 and, once again after having tuned the number of parameters, the inference time is 6ms. Proving, that it is possible to run state-of-the-art neural networks on small, CPU-constrained devices.

This work is a proof of concept for nano drone racing, further work on the perception part of the project should focus on one of the biggest limitations of the CNN, namely the sensitivity to white pixels. This issue arises from a dataset that mostly represents the gate on a black background, to facilitate the self-labeling algorithm. Moreover, deploying Dronet [12] on the AI-Deck will be beneficial considering it outperformed Gatenet in terms of accuracy and inference time. Another aspect that should be researched could be to include the skew of the gate in the prediction of the network to be able to extract the relative yaw between the gate and the drone.

With regards to the control side of the project, further steps could be focused on a better model of the drone to fix the attitude rates responses. To fix the thrust two approaches can be carried out, the first one is to modify the model such that during the training phase the maximum thrust varies thus leading to a network that can adapt to changes in maximum thrust. The second approach could be to implement a PID controller on the thrust command, similar to what is done for the attitude rates. Another interesting path to take would be to improve the attitude rate PID controller of the Crazyflie or substitute it with an INDI controller.

REFERENCES

[1] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, p. 604–609, Dec. 2020.

[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, p. 350–354, Nov. 2019.

[3] C. De Wagter, F. Paredes-Vallé, N. Sheth, and G. de Croon, "The sensing, state-estimation, and control behind the winning entry to the 2019 artificial intelligence robotic racing competition," *Field Robotics*, vol. 2, no. 1, p. 1263–1290, Mar. 2022.

[4] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 79767976, p. 982–987, Aug. 2023.

[5] L. Lamberti, V. Niculescu, M. Barciś, L. Bellone, E. Natalizio, L. Benini, and D. Palossi, "Tiny-pulp-dronets: Squeezing neural networks for faster and lighter inference on multi-tasking autonomous nano-drones," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Jun. 2022, p. 287–290.

[6] M. J. Anderson, J. G. Sullivan, J. L. Talley, K. M. Brink, S. B. Fuller, and T. L. Daniel, "The "smellicopter," a bio-hybrid odor localizing nano air vehicle," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 6077–6082.

[7] D. Palossi, N. Zimmerman, A. Burrello, F. Conti, H. Müller, L. M. Gambardella, L. Benini, A. Giusti, and J. Guzzi, "Fully onboard ai-powered human-drone pose estimation on ultra-low power autonomous flying nano-uavs," *IEEE Internet of Things Journal*, pp. 1–1, 2021.

[8] H. X. Pham, I. Bozcan, A. Sarabakha, S. Haddadin, and E. Kayacan, "Gatenet: An efficient deep neural network architecture for gate perception using fish-eye camera in autonomous drone racing," pp. 4176–4183, 2021.

[9] D. Scaramuzza and Z. Zhang, "Visual-inertial odometry of aerial robots," no. arXiv:1906.03289, Jun. 2019, arXiv:1906.03289 [cs]. [Online]. Available: http://arxiv.org/abs/1906.03289

[10] J. A. Cocoma-Ortega and J. Martínez-Carranza, "Towards high-speed localisation for autonomous drone racing," in *Advances in Soft Computing*, ser. Lecture Notes in Computer Science, L. Martínez-Villaseñor, I. Batyrshin, and A. Marín-Hernández, Eds. Cham: Springer International Publishing, 2019, p. 740–751.

[11] H. Nguyen, M. Kamel, K. Alexis, and R. Siegwart, "Model predictive control for micro aerial vehicles: A survey," no. arXiv:2011.11104, Nov. 2020, arXiv:2011.11104 [cs]. [Online]. Available: http://arxiv.org/abs/2011.11104

[12] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, p. 1088–1095, 4 2018.

[13] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar, and D. Scaramuzza, "Alphapilot: autonomous drone racing," *Autonomous Robots*, vol. 46, no. 1, p. 307–320, Jan. 2022, company: Springer Distributor: Springer Institution: Springer Label: Springer number: 1 publisher: Springer US Citiation Key: alphapilot.

[14] R. Ferede, C. De Wagter, G. de Croon, and D. Izzo, "End-to-end reinforcement learning for time-optimal quadcopter flight," in *ICRA 2024*, 2023.

[15] W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński, and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, 2017, pp. 37–42.

[16] "GAPflow," https://greenwaves-technologies.com/gapflow/, accessed: 2023-11-21.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[18] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," *CoRR*, vol. abs/1602.04938, 2016. [Online]. Available: http://arxiv.org/abs/1602.04938

Part II

Addition results

2

Addition results

This chapter describes additional experiments and results that were carried out during the project, which are not included in the scientific article. This might be useful if someone would like to continue working on this project or for the sake of completeness. The topics covered are the selection of the datasets, in section 2.1, next a description of the training pipeline in section 2.2, an evaluation of the relative position algorithm, section 2.3, and finally an analysis of the cost function for the guidance and control network, section 2.4.

2.1. Vision Datasets

For any network trained with supervised learning, datasets are crucial. A dataset that is not accurate, diverse, or with enough depth will surely result in a network that does not do the required task. Unfortunately, with datasets finding the right one is mostly trial and error. For this project, three different datasets were used: the Mavlab gates, the Simulated Gates, and the self-recorded dataset. In the following sections, each dataset will be described and the performance of the network will be evaluated.

2.1.1. Mavlab Gates

The initial dataset comprises in a collection of approximately 5,000 high-resolution images of gates captured by various high-resolution cameras across multiple drone racing circuits, during the 2019 Alphapilot challenge. This collection was hand-labeled and contributed significantly to the victorious outcome for the team in the said competition [2]. Representative samples from this dataset are exhibited in Figure 2.1. Concurrently, Figure 2.2 shows the visual performance of the Gatenet neural network post-training on this dataset. The efficacy of the network is compromised by several factors. First of all, the presence of multiple gates within a single image. The network's architecture can only detect one, during training, if multiple gates are present, the algorithm preferentially processes the largest. This penalizes the network for correct identifications. A secondary factor impeding performance is the diversity in luminosity and lighting conditions observed in the gate images. For instance, occasional blue LED illumination surrounding the gates, which suggests that the current network parameters may be insufficient to understand such variability. Lastly, a 'reality gap' exists, attributable to the discrepancy between the dataset images and the physical gates to be navigated during flight, predominantly coming from the low image quality captured by the Ai-Deck camera, as illustrated in figure 2.5.

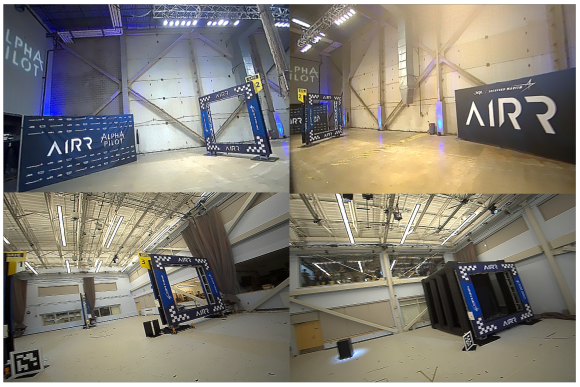


Figure 2.1: Examples of images from the Mavlab dataset for autonomous drone racing.

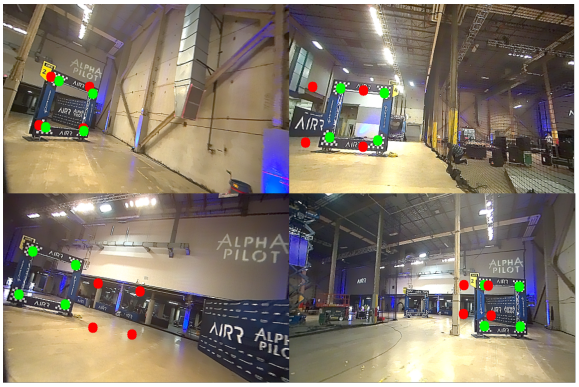


Figure 2.2: Results of GateNet on a test set of the Mavlab dataset

2.1.2. Simulated Gates

Given the limitations presented by the Mavlab dataset and its consequent impact on performance, questions arose regarding the network's capacity to effectively learn from images of such small resolutions. This concern prompted a shift to a simulated dataset comprising approximately 5000 images, as depicted in figure 2.3, utilized by Pham et al.[8]. This dataset is characterized by each image containing a single gate, with uniform brightness levels, thereby simplifying the task for the Convolutional Neural Network (CNN). The simplified conditions of the dataset allowed both the Gatenet and Dronet architectures to successfully learn from input images with the resolution of $[162 \times 122]$, as showcased in figure 2.4. Consequently, this dataset was selected to benchmark the performance of both networks within the scope of the study.

Figure 2.3: Simulation dataset used for training

Figure 2.4: Network results on the simulation dataset

2.1.3. Self Recorded Data

The difference between the simulated dataset and actual flight data remains too significant for networks to generate useful outcomes. In response, a new dataset was generated using the Hi-max color camera, a collection of 300 images, showcased in figure 2.5. These images were manually annotated. Upon training a network with this data, the task proved to be overly complex. The camera's deficient color and exposure quality make gate detection challenging even for human observers, which suggests that a lightweight network would also struggle with this task, as depicted in figure 2.6. In addition, another problem with this dataset was the tedious task of labeling the images which under the time constraints of this project forced us to change strategy and find a way to automatically label images using standard computer vision algorithms, eg. white pixel detection. Further insights from this dataset indicate that color does not significantly enhance the information for the task at hand, and the additional inference time required for the two extra color channels and the conversion from a Bayer pattern to RGB is considerable. Therefore, the focus shifted to using grayscale images.

To reduce complexity, a more straightforward and larger gate with distinct patterns was printed. Subsequently, a new dataset was captured with the HI-Max grayscale camera, as demonstrated in figure 2.7. This new dataset is composed of 2000 images, 400 of which were manually labeled, with

the remaining being processed by an automatic labeling tool. This tool calculates the gate's center and dimensions by pinpointing the white pixels within the image. A notable limitation of this method is the necessity for the gate to contrast against a dark background. Nonetheless, the network trained with this dataset yielded encouraging results, as evident in figures 2.8. Consequently, this network has been implemented on the AI-Deck for onboard gate detection tasks.

Figure 2.5: Results of GateNet on a test set of the Mavlab dataset

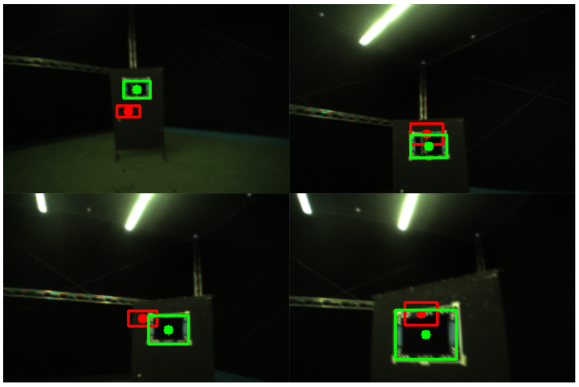


Figure 2.6: Results of GateNet on a test set of the Mavlab dataset

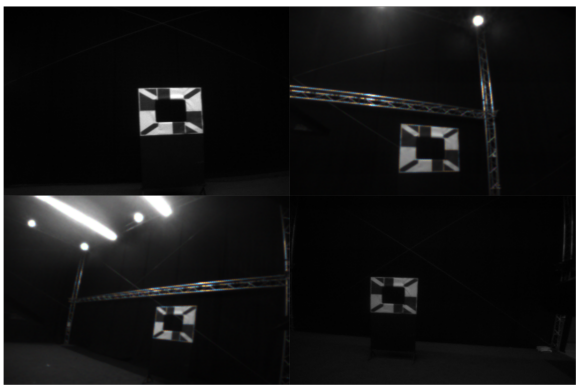


Figure 2.7: The newly designed gate used for the dataset

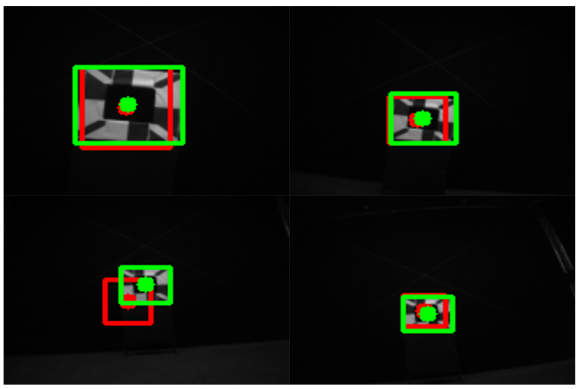


Figure 2.8: Performance results using the new gate design

2.2. Training Vision Networks

The training pipeline of the vision networks is divided into two phases, namely the data augmentation and the training phase.

First, the dataset is loaded, and the images are augmented to give more robustness to the network, the hue, saturation, and brightness are modified. Next, some blur is applied to simulate motion in the image. Finally, the image is shifted horizontally or vertically of a random number of pixels between zero and twenty, consequentially the labels are also changed.

Next, the network is trained normally for a certain amount of epochs. Following this, the network is pruned by a factor ranging between 20% and 50%, this step was later omitted. Finally, the network is quantized and trained at the same time, using a quantization-aware training framework of tensorflow[4].

2.2.1. Sparsifiacation

During the training, the network undergoes a sparsification process aimed at enhancing generalization and reducing inference time on the AI-Deck. The data in Table 2.1 compares the inference times of various network sizes, both sparsified and dense, deployed on the AI-Deck. The results indicate that this process does not lead to a reduction in inference time, nor does it yields a definitive enhancement in terms of loss. This absence is performance increase is due to the lack of software support for sparsity, namely, they do not support sparse matrix operations. Another reason is the fact that the GAP8 is

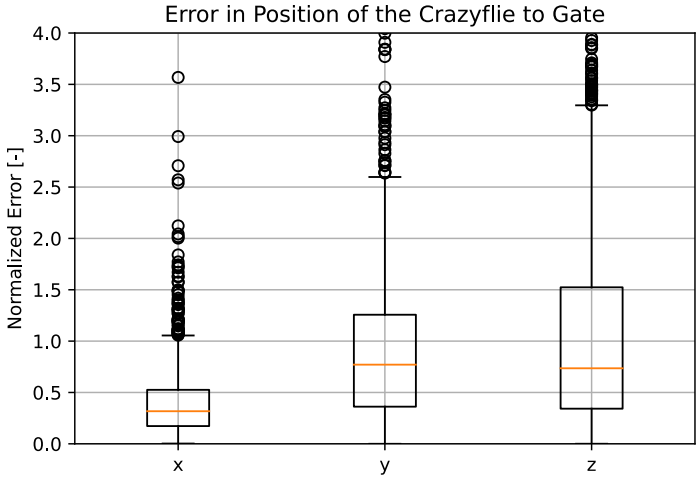


Figure 2.9: Normalized error between the estimated and true position across multiple flights, assuming a body frame with z down. Normalized means relative error divided by true distance.

designed to leverage parallelism of its eight cores. The sparsifiacation of the matrix could not be evenly distributed between the cores leading to some being underutilized. Due to these results, the sparsifiacation was discarded from the training pipeline.

Input Resolution	Base Inference Time [ms]	Sparsifed Inference Time [ms]
120x160	24	25
90x60	13	15
60x40	6	6

Table 2.1: Inference time of different input resolution networks on the AI-Deck, comparison between a dense network and a network sparsified at 50%

2.3. Relative Position Analysis

Figure 2.9 illustrated a collection of ten flights with vision net and relative position estimating the distance to the gate. Compared to the actual distance measured by the optitrack. The error is between 0.5m and 0.2m for the x estimate with a median of 0.4m. While y and z have larger errors, as they depend on the x axis prediction. However, it is important to mention that during testing the x direction was varied the most. Overall with this prediction, it is possible to navigate with the described pipeline(network plus relative position algorithm) but it is not very robust, as can be seen by the large amount of outliers.

2.4. Reinforcement learning cost function

Training a network with reinforcement learning is challenging because there is no control over the data that is being fed to the network, essentially the only parameters we can change are related to training, the most important of which is the reward function. Initially, the reward function used was the one developed by Ferede et al. [3] described in equation 2.1. This resulted in a flight that completed three laps but was not very stable. Figure 2.11a shows the trajectory of this flight, some loops on itself were present signaling a loss of control. Especially, if we compare this to the simulated trajectory, illustrated in figure 2.11b.

$$r(k) = \begin{cases} +10 - 10||\mathbf{p}_k - \mathbf{p}_{g_k}||, & \text{if gate passed} \\ -10, & \text{if collision} \\ ||\mathbf{p}_k - \mathbf{p}_{g_k}|| - ||\mathbf{p}_{k-1} - \mathbf{p}_{g_k}|| & \text{otherwise} \end{cases} \tag{2.1}$$

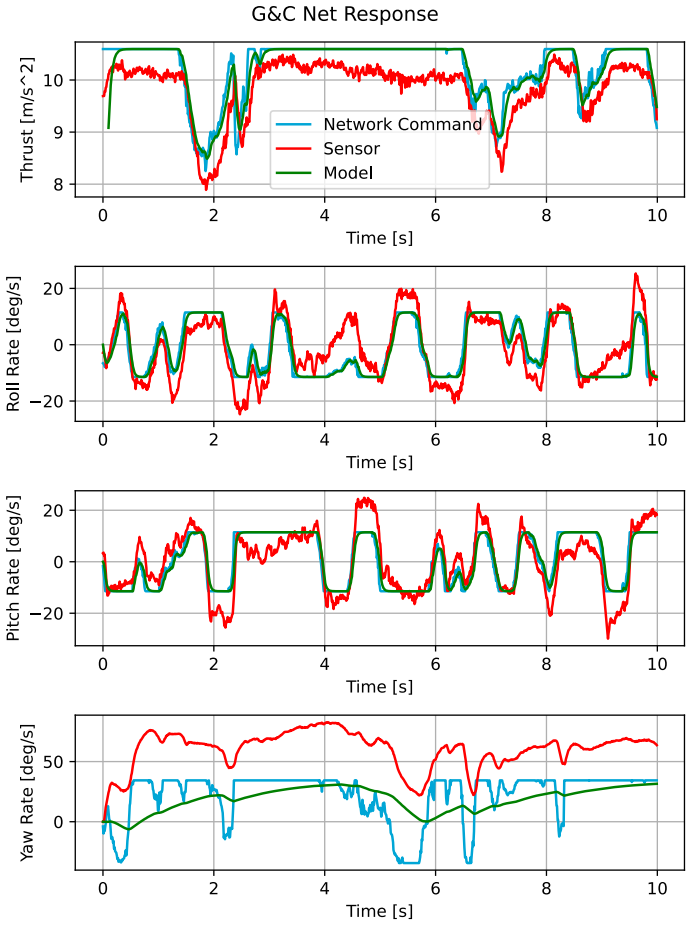
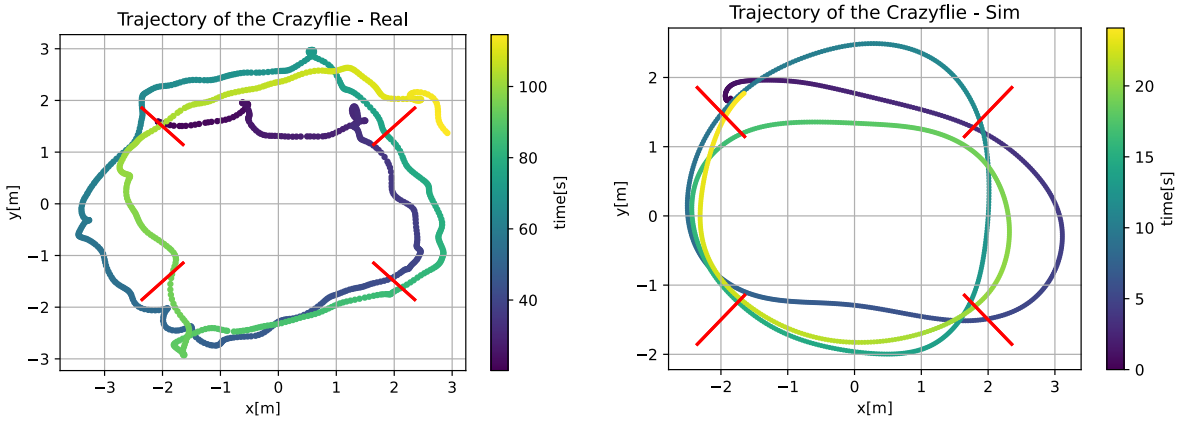


Figure 2.10: Response from the first iteration of the network

Digging deeper into the analysis of this flight, the network command has been mapped against the related sensor reading and the model of the drone, seen in figure 2.10. Immediately we can see the very poor performance of the yaw, in fact, two problems are present here. The first one is that the network constantly commands a positive yaw, which signifies that it did not learn to control the yaw. From this problem, we decided to introduce a yaw penalty every time the yaw of the drone is not aligned with the yaw of the gate. The second issue is that the attitude rate PID control on the crazyflie is not very effective as it is not able to match the commanded output yaw rate. Actually, throughout four test flights, it was determined that the controller at a bias of $35 deg/s$ which is subtracted from the network command would fix this error. In addition, from this flight, we can notice that the thrust undershoots especially at high speeds, and the pitch and roll overshoot.



(a) Real flight trajectory

(b) Simulated flight trajectory

Figure 2.11: Comparison of real and simulated flight trajectories

Part III

Literature review

Abstract

This study explores the intersection of autonomous nano-drone racing and neural networks, highlighting its potential in advancing drone technology. Autonomous drone racing serves as a platform for testing and improving the capabilities of autonomous systems, with applications in search and rescue operations and autonomous inspections. The software stack of autonomous nano-drone racing is divided into three parts: perception, control, and network acceleration. Neural networks prove to be the most accurate strategy for gate detection, but their processing time poses a challenge. A section-by-section analysis approach finds the bottleneck to be the first convolutional layer, 45% of the total computational time. The study also identifies useful planning and control methods, with the G&C Net as a prominent solution for providing motors rpm based on relative waypoints. Network acceleration methods, such as quantization and sparsification, are introduced to enhance speed and compression in the neural network.

3

Introduction

Autonomous drone racing is a cutting-edge field that combines drone racing with the technological advancements of autonomous systems. In recent years, this field has gained attention as a platform for research and development, due to its ability to provide a challenging environment to push drones to its limits and improve the capabilities of autonomous systems. These advancements can then be applied to the real world, in systems that need to navigate in complex environments, like search and rescue scenarios or autonomous inspection[Lit16]. For example, in earthquake or flood scenarios, drones must be able to quickly navigate through indoor environments without the use of GPS, relying only on onboard vision.

Neural Networks, a type of machine learning algorithm inspired by the human brain, are state-of-the-art right now for both control and perception applications in drone racing. However, deploying a neural network on edge devices, such as nano drones, is very challenging, due to several different reasons. Firstly, edge devices have limited computational resources. Secondly, edge devices have very strict latency requirements, especially drones as they need to fly at high speeds.

The intersection of autonomous drone racing and neural networks presents a compelling road for research. In this thesis project, the goal is to deploy a perception neural network on a Crazyflie equipped with AI-Deck[Lit3] for gate detection. Followed by the deployment of a planning and control neural network on the same drone. Due to the small dimensions of the drone, the processing power is very limited. Therefore, traditional neural networks cannot be used, as they are normally deployed on powerful Graphics Processing Units(GPU).

Autonomous nano-drone racing is a field that has not been explored and pushing the limits of the Crazyflie drone equipped with the AI-Deck is an important step in this field. There are plenty of challenges on both the control and perception side, leading to a vast knowledge gap[Lit28][Lit26]. Consequentially, from this gap, the main research question follows:

How can state of the art perception and control algorithms for autonomous drone racing be simplified to fit nano-drones, specifically a crazyflie equipped with AI-Deck?

This question is very general and therefore it needs to be divided into sub-questions according to the literature study. The first one is:

Which perception algorithms perform best in the context of gate detection for autonomous drone racing?

This question can be broken down in two distinct parts which together will lead to a well-rounded answer. The first is the "perception algorithms", which refers to the methods that include both learning-based and traditional algorithms. Next, "perform best", the performance of the perception algorithms "in context of gate detection for drone racing" can be determined with the accuracy with which gate are detected. Now for the control section of the thesis:

Which control algorithms perform best for high speed autonomous drone racing?

This question is asking to determine which control algorithms excel in the context of high-speed autonomous nano-drone racing. These algorithms can once again be both traditional or learning based. For both of these questions, it is implied that the best algorithm can be accelerated to maximise

performance as, at the end of the day, the goal is to fly autonomously as fast as possible with a specific nano-drone running all software onboard, without external positioning systems.

The purpose of this report is to investigate these research questions, determine the state of the art algorithms and predict problems that could be encountered during the research. For this reason, the research presents state-of-the-art strategies for each stage of the software stack. Starting from perception in chapter 5, going from image to position and state, where traditional and learning-based strategies are presented. Moving on to the control module in chapter 6, this module is in charge of going from states to motor rpm. Also here, traditional and learning-based approaches are presented. Next, chapter 7 treats methods to accelerate the deployment of neural network to reduce latency. Following this, chapter 8 lays out the plan of this project with the use of a gantt chart. Finally, chapter 9 presents the first results of this project.

4

Hardware Platform

There are many different types of drone configurations and sizes, this project deals with nano-drones. These are a category of unmanned areal vehicles with a wingspan between 1.5cm and 15cm and a weight between 3g and 50g. Being of such a small size these drones are incredibly agile and maneuverable, which allows for tight space navigation, this makes them valuable in search and rescue scenarios, monitoring, inspections, surveillance, and gas source seeking. The selected model for this project is the CrazyFlie 2.1, shown in Figure 4.1 due to its accessibility and the versatile open-source development platform. The main processor is a STM32F405(Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash), and radio and power management is done by nRF51822(Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash). The platform also has 3 axis accelerometers/gyroscopes and a high-precision pressure sensor.[1][Lit3]

This setup does not have sufficient computational power to run both drone control algorithms and an image processing neural network onboard in real-time. Fortunately, recent advancements in microprocessors optimise the inference speed of neural network. The low-power GAP8 processor developed by GreenWave Technologies is a great example of this. The GAP8 incorporates nine RISC-V cores capable of running at up to 250 MHz along with a neural processor designed to accelerate convolutional neural networks. It can offer a peak performance of up to 200 MOPS at 1 mW and up to 10 GOPS at a few tens of mW. BitCraze integrated this microprocessor on the AI-Deck along with a Wi-Fi model and an RBG camera. It features 64kB level 1 memory and 512kB level 2 memory. The design of the AI-Deck is based on the PULP-shield[Lit3].

Figure 4.1: CrazyFlie2.1 illustration

[1]https://store.bitcraze.io/collections/kits/products/crazyflie-2-1

5
Perception

The perception section of the software stack estimates the vehicle state and perceives the environment using onboard sensors. The most common approaches for accurate perception are GPS-based Odometry, LIDAR-based Odometry and Visual-Inertial Odometry. This project focuses on vision-based drone racing with nano-drones making the first two options irrelevant. This chapter focuses only on Visual-Inertial Odometry, using a camera and the sensor readings. Starting with a description of this approach in section 5.1 in particular how measurements from different sensors are processed. Next, the PnP algorithm, to transform 2D coordinates to 3D, is explained, section 5.2. Gate Detection algorithms are then explained in section 5.3 and section 5.4. And traded off in section 5.5. Finally, the software stack is explained in section 5.6

5.1. Visual-Inertial Odometry

Visual-Inertial Odometry(VIO) is the process of estimating the state, which consists of pose(position relative to a landmark) and velocity of a drone, by using only onboard cameras(or a single monocular camera in the case of a crazyflie) and Inertial Measurement Units(IMUs), as shown in Figure 5.1. It is a possible alternative in the absence of GPS but can also be used to complement it.[Lit41]

The two sensors, camera and IMUs, augment each other very well. A camera takes a snapshot of a scene and converts it to a 2D image, very useful for perception tasks like recognising where you are in a scene. Although, it has various limitations namely, the slow output rate(80Hz), the scale ambiguity due to monocular effects, motion blur, High Dynamic Range(over- and under-exposure), and most importantly the time it takes to process an image. An IMU is an electronic device that measures a body's specific forces(then converted to accelerations) and angular rates. All its measurements are done in the IMU frame(which is the same as the body frame in most cases) rendering this device scene-independent. IMUs have a very high output rate(1000Hz) leading to better measurements at high speeds. The one big disadvantage is that it suffers from sensor biases leading to a loss of accuracy over time that needs to be corrected. Furthermore, the integration of IMU measurements from acceleration to velocity introduces challenges, as integrating biases amplifies them. For this reason, to achieve accurate and robust state estimation IMUs and cameras need to be combined.[Lit41] The camera is used for relative position and orientation, as the 3D environment is represented as a set of landmarks projected to 2D image coordinates. The IMU measures the angular velocity and external acceleration, following Equation 5.1. Where subscript I refers to the IMU frame and W in the world frame, b, n are biases.[Lit41]

$$\boldsymbol{\omega} = \boldsymbol{\omega}_I + \mathbf{b} + \mathbf{n}, \quad \mathbf{a} = \mathbf{R}_{IW}\left(\mathbf{a}_W - \mathbf{g}_W\right) + \mathbf{b} + \mathbf{n} \tag{5.1}$$

In VIO, at each point in time, the orientation with regards to the earth of the IMU needs to be known, in order to remove gravity from the measurements. In addition to this, the velocity at which the IMU is moving and biases need to be estimated. The first one is needed to integrate acceleration to get the position. The latter is needed for computing actual sensor angular velocity and acceleration from raw measurements. Thus, for high-quality measurements, the bias needs to be approximated correctly.

There are two main approaches to fuse vision and IMU data: loosely coupled and tightly coupled. The former computes two independent state estimates one from vision measurements and the other

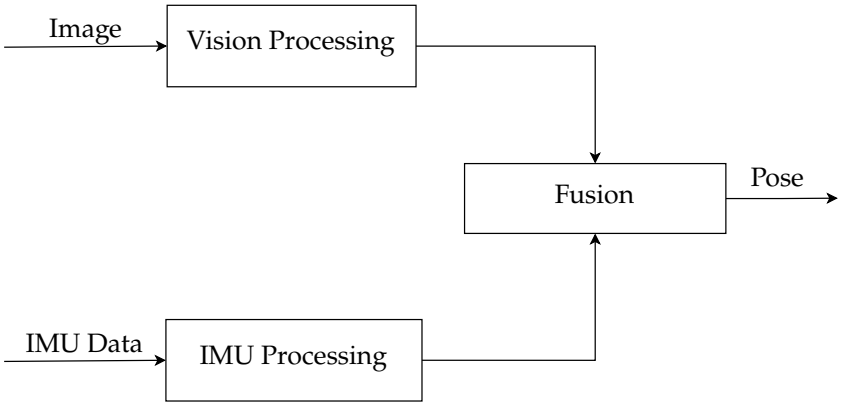


Figure 5.1: VIO Flow diagram

from inertial measurements. These estimates are then fused to get a final output. The latter, generally more accurate, computes the output directly from the raw data of cameras and IMUs together.

Diving deeper into tightly coupled VIO, three categories can be identified, which trade-off between accuracy and computational demand: filtering, fixed-lag smoothing and full smoothing.

- **Filtering algorithms** only estimate the latest state allowing efficient computations. Most methods use an Extended Kalman Filter, where they propagate the state of the system with inertial measurements and perform the update with the camera measurements. One of the first and most adopted methods is the multi-state constraint Kalman filter[Lit33]. There are two main errors in filtering approaches: information from older states is absorbed, and linearisation errors can cause a loss of accuracy.
- **Fixed-lag smoothing methods** estimate the states within a given time window while marginalising older states. This leads to a more accurate estimate as it relinearises past measurements. These approaches are also more robust to outliers. However, since the marginalise measurements they still have inconsistencies and linearisation errors. [Lit17]
- **Full smoothing methods** estimate the entire history of the states by solving a large nonlinear problem. These methods have the highest accuracy but the biggest computational load, and solving this in real-time is not feasible, especially for long flights. For this reason, it is not a very common approach. [Lit25]

For a nano drone with very limited computational capabilities, fixed-lag smoothing methods and full smoothing methods are not feasible, therefore the next Section, 5.1.1 focuses on the most common type of filtering, namely the Extended Kalman Filter, and new solution for sensor fusion named Visual model-predictive localization, Section 5.1.2.

5.1.1. Extended Kalman Filter

The Extended Kalman Filter(EKF), for this project, is useful for the estimation of the position of a drone relative to at least three landmarks. This algorithm is carried out in two steps namely, prediction and update. The prediction step produces a predicted location based on the previous location and IMU data. The update step then fuses the visual data to produce an overall estimate. As mentioned earlier, the camera and IMU data update at different rates, requiring the utilisation of an asynchronous updating technique in the algorithm. This technique ensures that the update step is executed only when new vision data becomes available.

In an EKF, the state and observation equation, represented in Equation 5.2 do not need to be linear, on the contrary of a normal Kalman filter, but they just need to be differentiable. The state vector, x_k, contains the predicted coordinates so it can be position and velocity measurements or quaternions. While, the measurement vector z_k consists of the measurement from the sensors, relative position coordinates from the camera and IMUs.

$$\boldsymbol{x}_k = f\left(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k\right) + \boldsymbol{w}_k \tag{5.2a}$$

$$\boldsymbol{z}_k = h\left(\boldsymbol{x}_k\right) + \boldsymbol{v}_k \tag{5.2b}$$

The prediction step, shown in Equation 5.3, uses the system model as well as the IMU measurements, $\mathbf{u}_k$ in this case, to predict the state vector, $\hat{\boldsymbol{x}}_{k|k-1}$, and the covariance matrix, $\boldsymbol{P}_{k|k-1}$.

$$\hat{\boldsymbol{x}}_{k|k-1} = f\left(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k\right) \tag{5.3a}$$

$$\boldsymbol{P}_{k|k-1} = \boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^T + \boldsymbol{Q}_k \tag{5.3b}$$

The next step in the EKF is the update step. Shown in Equation 5.4, the measurement model is used to update the predicted state using measurements from a camera, $\mathbf{z}_k$. This step also updates the residual covariance ($\boldsymbol{S}_k$), the Kalman gain ($\boldsymbol{K}_k$) and the covariance estimate($\boldsymbol{P}_{k|k}$). In order to have a better prediction at the next iteration.

$$\tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - h\left(\hat{\boldsymbol{x}}_{k|k-1}\right) \tag{5.4a}$$

$$\boldsymbol{S}_k = \boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k \tag{5.4b}$$

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T \boldsymbol{S}_k^{-1} \tag{5.4c}$$

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k \tag{5.4d}$$

$$\boldsymbol{P}_{k|k} = \left(\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k\right) \boldsymbol{P}_{k|k-1} \tag{5.4e}$$

5.1.2. Visual model-predictive localisation

In [Lit26] the authors propose a novel sensor fusion method called visual model-predictive localisation. The main idea behind this approach is to rely as much as possible on a predictive model of the drone dynamics, while correcting the model by localising the drone visually based on the detected gates and their supposed positions on the global map. Within a small time window, visual model-predictive localisation approximates the error between the model prediction position and the visual measurements as a linear function. Once the parameters of this function are estimated by a Random Sample Consensus(RANSAC) algorithm, this error model can be used to update the prediction.

In other words, visual model-predictive localisation uses a combination of predictive modeling and visual sensing to enable fast and accurate flight in an autonomous racing drone. By using a predictive model of the drone's dynamics, visual model-predictive localisation can anticipate how it will move through space and adjust its flight path accordingly. At the same time, by using visual sensing to detect gates and other landmarks in its environment, visual model-predictive localisation can correct for any errors or deviations from its predicted path. This allows it to fly quickly and accurately through complex courses with changing obstacles. This method gives unbiased estimations of the parameters but it is prone to error if there are outliers in the analysed time window. For this reason, a RANSAC algorithm is employed. This is an iterative method to estimate parameters from a set of data that contains outliers. It works in 3 steps:

1. Randomly select a minimal subset of data points that are assumed to be inliers and use them to fit a model.
2. Determine the remaining data points that are consistent with the fitted model based on a predefined threshold. These data points are called inliers.
3. If the number of inliers is greater than a predefined threshold, re-estimate the model using all of the inliers and terminate the algorithm. Otherwise, repeat step 1 until the maximum number of iterations is reached.

5.2. Perspective-n-Point

Once features, such as gates, have been detected on a 2D projection of the image, the next problem is estimating the pose of the calibrated camera, which is on the drone. A drone has 6 degrees-of-freedom that are 3 rotations and 3 translations [Lit42].

The PnP algorithm requires at least three correspondences between the 3D object and the 2D image, i.e., three pairs of points in the image and the object with known correspondence. In practice, more correspondences are used to improve the accuracy of the estimation. Assuming only three points are known in the world coordinate system($\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$)and their corresponding 2D projections on a camera. The first step in a P3P algorithm is to set the camera's intrinsic parameters, such as focal length and

principal point[Lit42]. Next, the camera orientation and position are solved for according to Equation 5.5, where **C** is the center of the camera, **R** is the rotation matrix that maps the world coordinate system to the camera coordinate system, and **e** are the basis vector of the world coordinate systems. Here the unknowns that are solved for are **R** and **C**.

$$\mathbf{d}_1 = \frac{\mathbf{P}_1 - \mathbf{C}}{|\mathbf{P}_1 - \mathbf{C}|} = \mathbf{R} \cdot \mathbf{e}_1 \tag{5.5a}$$

$$\mathbf{d}_2 = \frac{\mathbf{P}_2 - \mathbf{C}}{|\mathbf{P}_2 - \mathbf{C}|} = \mathbf{R} \cdot \mathbf{e}_2 \tag{5.5b}$$

$$\mathbf{d}_3 = \frac{\mathbf{P}_3 - \mathbf{C}}{|\mathbf{P}_3 - \mathbf{C}|} = \mathbf{R} \cdot \mathbf{e}_3 \tag{5.5c}$$

Finally, the translation vector **t** can be determined according to Equation 5.6. Knowing, both rotation and translation the camera pose can be extracted. There are multiple other methods to solve the PnP problem and quite a few efficient off the shelf solution, in libraries like OpenCV[1] or OpenMVG[2].

$$\mathbf{t} = -\mathbf{RC} \tag{5.6}$$

5.3. Traditional Gate Perception

Gate detection can be accomplished by different computer vision algorithms like Viola and Jones and Hough Transforms. However, these approaches do not perform well on drones especially due to the lack of rotation invariance. Table 5.1 presents an overview of gate detection methods both traditional and learning based with a high-level selection. From this table, the only traditional approach selected was snake gate [Lit27]. This approach works by searching for gates based on their colours. The search starts by picking a random point on the image ($\mathbf{P}_0$). If the random point is of the correct colour, the search follows pixels of the same colour "up and down" to find points, $\mathbf{P}_1$ and $\mathbf{P}_2$. These two points are then used as a starting point of a "left right" search to find $\mathbf{P}_3$ and $\mathbf{P}_4$. In order to avoid detection of small colour blocks the same colour as the gate, a minimum distance threshold is introduced in both the lateral and vertical searches following Equation 5.7. If the gate's image is continuous and the gate's edges are smooth then this algorithm should detect all four corners. However, varying light conditions can jeopardise the results. Figure 5.2 is an example of how the algorithm would work. In most cases, snake gate detects more gates than the number of real gates in the image. This is due to the fact that there will be duplicates on the same gate. This does not influence performance as they are true positive detections. False positive detections of gate can influence performance these are solved by adding another threshold, Equation 5.8. Where N_c is the number of pixels in the polygon with the target colour. Thus, if not enough pixels of the target colour are detected then the gate is ignored.

$$|\mathbf{P}_1 - \mathbf{P}_2| < \sigma_L \tag{5.7}$$

$$\sigma_{cf} < cf = \frac{N_c}{N} \tag{5.8}$$

5.4. Learning-Based Gate Perception

Over the last years neural networks have gained popularity due to their ability to cope with different types of input data, their accuracy and the simplification of deployment on hardware. The goal of a perception neural network is to detect landmarks from an image, eg. gate locations in the case of drone racing. Referring back to Figure 5.1, perception networks perform extremely well in vision processing.Table 5.1 presents a high-level selection of gate detection methods. A few computationally light networks are selected and their principles are explained in the following sections subsection 5.4.1, subsection 5.4.2 and subsection 5.4.3.

[1]https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html
[2]https://github.com/openMVG/openMVG

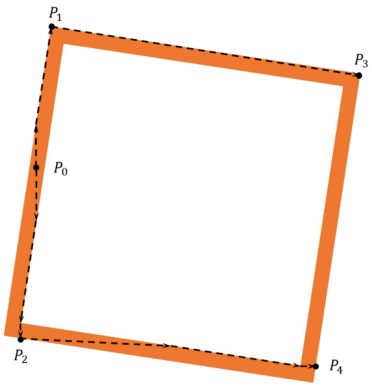


Figure 5.2: Example of snake gate detection[Lit27]

5.4.1. Gate detection by semantic segmentation

The network in [Lit46], it is a fully convolutional deep neural network that consists of a four-level U-Net. Its objective is to convert every input image into a binary mask that segments all the gates in sight, as shown in Figure 5.3. A U-Net is a type of Convolutional Neural Network(CNN) used for image segmentation. It has a symmetric encoder-decoder architecture with skip connections between the encoder and decoder paths. These skip connections enable the network to effectively learn spatial information from a given input image, allowing it to accurately segment objects in the image [Lit40]. In this case the network has $[64, 128, 256, 256]$ convolutional filters of size 3×3 and elementwise sum skip connections. The most challenging part of this neural network is creating the dataset. This net was trained on 2336 images with the binary mask manually annotated, leading to a supervised training method.

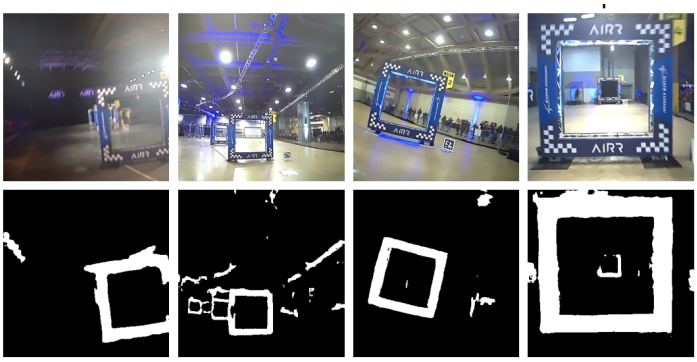


Figure 5.3: Gate detection by semantic segmentation input image and corresponding binary mask[Lit46]

5.4.2. GateNet

GateNet [Lit34] is a novel network that from a single image predicts center, orientation and distance of gates. It is also designed to run high-speed inference on the Nvidia Jetson TX2, up to 60Hz. The architecture of the network is a CNN with a single fully connected layer at the end. The convolutional part extracts the features from the image and the fully connected layers predicts the confidence values, orientation and distance of the gates. As noticeable, in Figure 5.4, this network has six convolutional layers, the first five of which are followed by batch normalisation, rectified linear unit activation and max pooling. The last convolutional layer is not followed by the max pooling. The convolution layers condense the information in the images to $3 \times 5 \times 16$ to reduce computational load, this image is then flattened and a dense layer is applied. To finally output the five aforementioned predictions, in a reshaped form, $4 \times 3 \times 5$, where each cell in the grid contains the predicted values of $\{x, y, d, \theta, c\}$. Where x and y are the normalised pixel differences between the top-left corner of the the grid and the gates center, d is the relative distance to the gates center in meters, θ is the relative orientation of the

Algorithm	Selected	Reason
Snake Gate[Lit27]	Yes	Extremely efficient and proven to work on nano-drones.
VINS-Mono[Lit37]	No	Computationally expensive and complex to deploy with gates.
DroNet [Lit30]	Yes	Computationally cheap, robust and adaptable.
GateNet [Lit34]	Yes	Computationally cheap, robust and adaptable.
Gate Detection by Semantic Segmentation[Lit46]	Yes	Very robust and adaptable, proven flight experience
AplhaPilot [Lit9]	No	Robust but computationally expensive and less performant than the previous approach.
Deep Pilot [Lit39]	No	End-to-end drone navigation, complications will arise in the control of the nano-drone.
Sparse Gated Recurrent Network[Lit1]	No	Event based vision.
Deep Patch Visual Odometry [Lit45]	No	Network applied to VIO not just gate detection.

Table 5.1: High-level gate detection algorithm selection

gate in radians, so the difference between the drone's yaw and the gate's yaw. Finally, c is the confidence value, which has a value of 1 if a gates center is located inside the image and 0 otherwise. The training was supervised on images generated on a simulator. The same authors of this paper developed a new network named PencilNet [Lit35], which has the same architecture and properties as GateNet but the input image has a pencil filter applied to it. By doing so, the network is more effective for simulation to reality transfer learning.

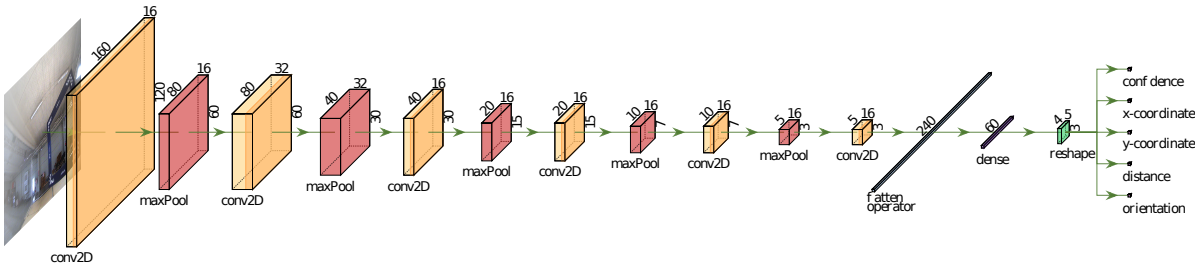


Figure 5.4: Architecture of GateNet network[Lit34]

5.4.3. DroNet

Another state-of-the-art end-to-end vision network is DroNet[Lit30]. A CNN whose initial purpose was to reliably fly an autonomous drone through the streets of a city. The architecture as shown in Figure 5.5, is a variant of the ResNet-8 DNN with three residual blocks. The purpose is obstacle avoidance thus, the outputs are a steering angle and a probability of collision. However, due to its efficient structure, many researchers used this network as a backbone feature extractor with a different output layer. For example, the authors in [Lit21] change the output layer with multi-layer perception to estimate the mean and variance of the gate's next pose with regards to the camera for a model predictive control to

then compute a trajectory for the drone. This network takes as input an 320×240 RBG image and the output is the mean and the variance of these values, $\tilde{\mathbf{z}} = [\tilde{r}, \tilde{\theta}, \tilde{\psi}, \tilde{\phi}]^\top \in \mathbb{R}^4$. The values are presented in spherical coordinates as it is advantageous to decouple distance from the image coordinates.

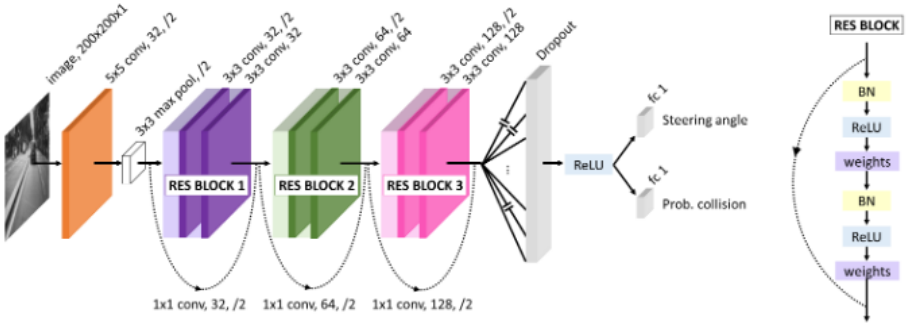


Figure 5.5: Architecture of Dronet network[Lit30]

5.5. Trade-off Vision Methods

In this chapter various topics have been covered namely, different sensor fusion methods, the PnP algorithm and different gate perception methods. This section performs a trade-off between the different ways to recognise a gate from an image, with the main parameters being:

- The computational efficiency of the algorithm.
- The accuracy of the output data.

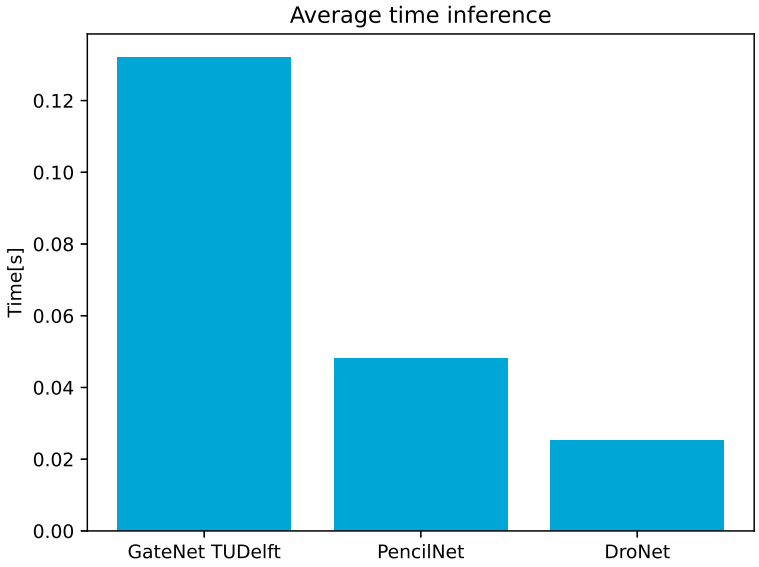


Figure 5.6: Inference of the Networks run on an Intel i5 core, single thread using the tflite inference engine

In recent years, gate detection has been a very researched topic but from literature, the choice was narrowed down to the four state-of-the-art methods described in the previous sections and shown in Table 5.2. To determine the efficiency of each network, inference was run on an Intel i5 core using a single thread and the tflite inference engine[3], the results are shown in Figure 5.6. From this graph, one

[3]https://www.tensorflow.org/lite

can notice that DroNet has the best inference time and the GateNet TU Delft, so the gate detection by semantic segmentation, required the most computational power. This is due to the fact that GateNet TU Delft has an Encoder and Decoder structure, requiring it to reconstruct the input image. It is important to note that for this benchmark PencilNet and DroNet were scaled up to have the same input size as GateNet TU Delft($3 \times 360 \times 360$). Moreover, when deploying these networks on the AI-Deck they will run on a RISC-V architecture, so a CPU-like chip, thus similar performance, relative to each other is expected. On a final note, the snake gate algorithm is not tested as it is clearly the most efficient algorithm, mainly because it is not a neural network.

The approaches have been graded in Table 5.2. Overall, the snake gate is very efficient but it only detects orange-coloured gates and is prone to errors due to false positives and motion blur. GateNet TU Delft is more accurate compared to the previous one but it outputs a mask of the gate requiring a PnP algorithm, also it is a very large network making it not suitable for the AI-Deck. GateNet and PencilNet have the same neural network architecture and are quite computationally efficient due to the low number of parameters(32K), the main drawback of these methods is that the output is in Cartesian coordinates of the image requiring and extra conversion step, similar to PnP. Finally, DroNet is more computationally efficient than the GateNet and with an output layer that can be adapted, making it the best option for gate recognition. These approaches will have to be tested on the AI-Deck to actually understand which one performs best.

ID	Title	Efficiency	Ouput Data	PnP	Overall
1	Snake Gate [Lit27]	9	2	Yes	11
2	Gate detection by semantic segmentation[Lit46]	3	8	Yes	11
3	Gatenet: An efficient deep neural network architecture for gate perception[Lit34]	7	6	Yes	13
4	PencilNet: Zero-Shot Sim-to-Real Transfer Learning for Robust Gate Perception[Lit35]	7	6	Yes	13
5	DroNet: Beauty and the Beast[Lit21]	8	7	No	15

Table 5.2: Trade-off of different vision algorithms

5.6. Software Stack

The perception module of the software stack needs to find out where the drone is relative to a gate and where it needs to go. Figure 5.7 shows a flow chart on how this is done in software. First, an image from a camera is fed to a neural network. The output of which should be the coordinates on the image of the 4 corners of a gate. These 2D coordinates are then transformed into 3D world coordinates using a PnP algorithm. Finally, the 3D coordinates are fed to an EFK which, together with IMU data, will output the relative position and state of the drone.

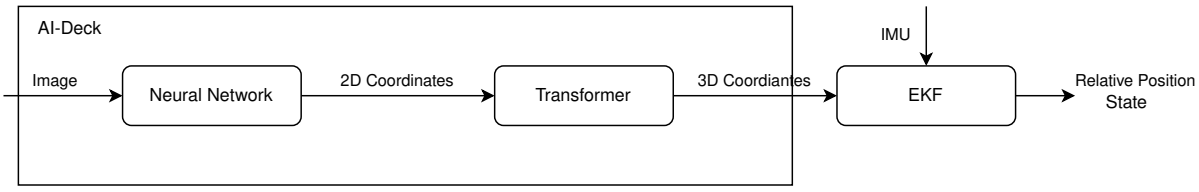


Figure 5.7: Perception Software Stack

6 Control

Once the state estimate has been obtained the next step is to plan a feasible trajectory, which predicts the future states of the drone such that a minimum lap time is reached without crashing, this is the planning phase. After this, the control module needs to make sure that this trajectory is followed by the drone. In this chapter, first of all, traditional control methods are covered, section 6.1, this includes both planning of trajectories and control of drones. Next, learning-based control approaches are described, section 6.2 methods which bypass the planning stage and convert sensors observations to actuators commands directly. Finally, a trade-off of all the methods is carried out in section 6.3.

6.1. Traditional Control Methods

Traditional control methods are separated in two parts: the planning, where an optimal trajectory is planned, and the control, where the optimal trajectory is followed. There are plenty of different approaches to this problem but, for the purpose of this report, the focus is on the most time optimal and least computationally expensive, as this is what is needed in drone racing.

6.1.1. Planning

As mentioned before, once a state estimate has been obtained, the next step is to plan a feasible, time-optimal trajectory, which keeps in account the limits of the drone as well as the constraints imposed by the environment. The planning methods can be categorised into four different methods namely, polynomial trajectory planning, optimisation-based planning, search-based planning, and sampling-based planning.

In polynomial trajectory planning methods, a polynomial is used to represent the trajectory and its coefficients are computed based on the specific constraints, such as initial and final position, waypoints, velocities and accelerations. The full-state trajectory planning is reduced to only 4 states(3D position and heading). By taking derivatives of these polynomials valuable data like velocity, acceleration, jerk and snap can be determined and used for optimisation purposes. A great example of this is in paper [Lit32], where the authors minimise the snap in a trajectory. They segment a trajectory in various keyframes and generate a trajectory that passes through these keyframes at a given time while staying in a predetermined safety corridor. Next the trajectory, is optimised for the 2nd derivative of acceleration(so 4th of position) and 2nd derivative of yaw.

Optimisation based trajectory planning starts with a selection of an objective function. Next, a solver is used to find the optimal sequence of states at every timestep, complying with the physical constraints of the drone. These approaches have been researched thoroughly using different models of the drone, such as point-mass models[Lit10], simplified quadrotor models, or full state models[Lit11]. For drone racing, another constrain for path planning are waypoints. The standard approach to this problem, carried out in [Lit4], is to generate an optimal continuous trajectory that passes through the sequence of waypoints using by solving the objective function with pseudo-spectral methods, setting an arrival time constraint at each waypoint.

Search-based planning methods generate trajectories through algorithms such as A^*, Dijkstra's, and probabilistic roadmap searches. Normally they consider velocity, acceleration, or jerk. The procedure

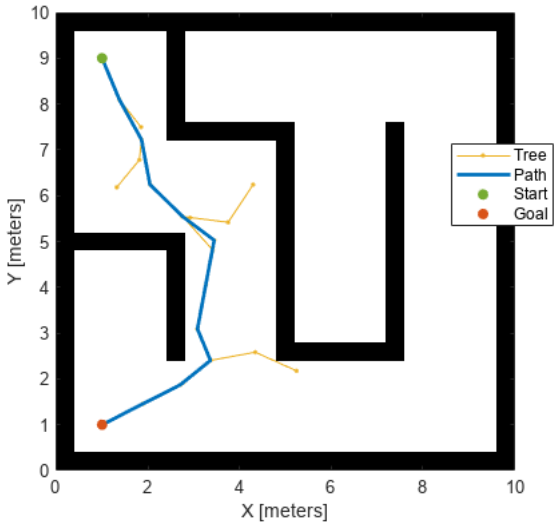


Figure 6.1: Simple example of rapidly exploring random tree[Lit31]

starts with generating a set of candidate trajectories and then evaluating them in terms of optimality and feasibility. These algorithms are extremely powerful and versatile but they are also computationally expensive. Therefore, not feasible to run on a nano drone.

Finally, sampling-based methods involves randomly sampling the configuration space of the drone and constructing a graph of connected samples to generate a collision-free path. One common approach is to use a graph-based representation of the configuration space, such as a rapidly-exploring random tree[Lit47]. In these methods, the randomly generated configurations are added as nodes to the graph, and edges are added between neighbouring nodes that do not result in collisions. The resulting graph represents a set of feasible trajectories that can be used to plan a path from the robot's current configuration to the goal configuration, as shown in Figure 6.1. Sampling-based trajectory planning is computationally efficient and can handle high-dimensional configuration spaces, but it may not always find the optimal trajectory.

Overall, optimisation based trajectories are the best option for drone racing due to their low computational cost and ability to plan 4-d trajectories, including time. Nevertheless, planning, in general, faces challenges due to inconsistencies in a drone's ability to adhere to the intended course, potentially resulting in system failure. Thus, in section 6.2, learning control methods are discussed where planning and control and coupled in closed-loop manner.

6.1.2. Control

Controllers make real-time decisions that guide the drone toward a destination. Normally, there are high-level controllers that output desired virtual inputs like velocity or body rates, and low-level controllers that control each individual motor based on the desired virtual input. There are many open-source implementations of low-level controllers like Paparazzi, PixHawk and Betaflight. For the purpose of this report, the focus is on BitCraze's implementation, seen in Figure 6.2. The workflow of the algorithm works as follows, a high-level controller sends a desired position to the Position PID controller. This outputs the roll and pitch rates that are then fed to the Attitude rate PID which results in the motor commands.

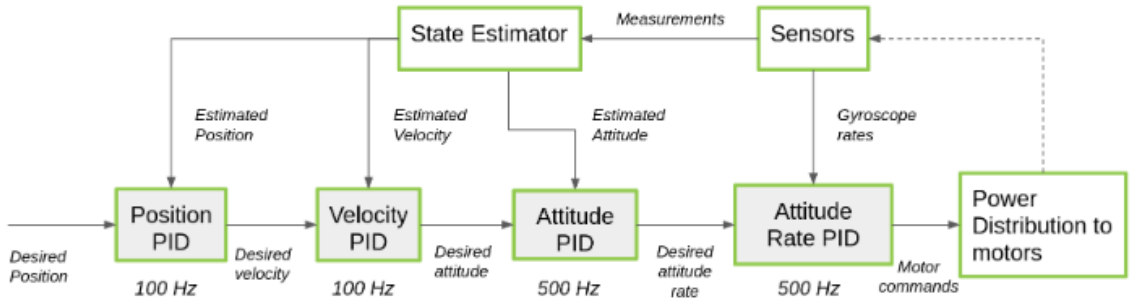


Figure 6.2: Cascade PID by bitcraze[Lit3]

Moving on to high-level control, the most common approach is Model-Based Control. Where, a model of the dynamic systems is used to compute the control commands to reach a certain objective. There are many methods that follow this approach the two major ones being Incremental Nonlinear Dynamic Inversion(INDI) and Model Predictive Control(MPC). INDI measures the angular acceleration and an increment of the control outputs is calculated based on the desired increment of angular acceleration. This controller is robust to modelling errors, disturbances, and can handle highly nonlinear systems. However, it can be computationally intensive. [Lit43]

MPC is a control strategy that uses a model of the system being controlled to predict its future behaviour and then computes a sequence of control actions that optimise a given objective function over a finite time horizon, subject to constraints on the system and the control inputs. This method exploits the full flight envelope of the drone achieving higher velocities and better performances. [Lit6]

6.2. Learning-Based Control Methods

As previously mentioned, learning-based control methods tend to merge planning and control, eliminating the need for high-level trajectory planning. This leads to more robust systems as sensor noise, model uncertainties and system latency can be included in the training data of the policy. Moreover, by eliminating the planning stage there are no more discrepancies between planning and control. For the purpose of this report, the learning-based approaches are divided in their groups based on the output they provide namely, linear velocity, total thrust and body rates and single rotor thrust.

Linear velocity approaches are control policies that specify high-level commands like waypoints or velocities. The main feature of this approach is that it does not take into account the vehicle dynamics, making it a method that can easily be transferred to different platforms. The drawback of this is that it will not exploit the full capabilities of the platform. An example of this method can be seen in [Lit2], where a network given a location computes an action that a drone needs to get to that location in terms of the velocity vector. In this specific instance at the desired location there was an object to pick up. Once the object was picked up the network adapts the different dynamics. It is important to note that the action computed is then carried out by an MPC.

Total thrust and body rates approaches output thrust and body rates allowing for more aggressive manoeuvres. This paper, [Lit22], presents an approach that combines IMU measurements, reference trajectory and feature tracks and output collective thrust and body rate commands. Its purpose is to autonomously perform three acrobatic manoeuvres namely, the Power Loop, the Barrel Roll and the Matty Flip. The structure of the network is shown in Figure 6.3. This approach does not need a full-state estimation but only requires inertial measurements to feedback control on the body rates.

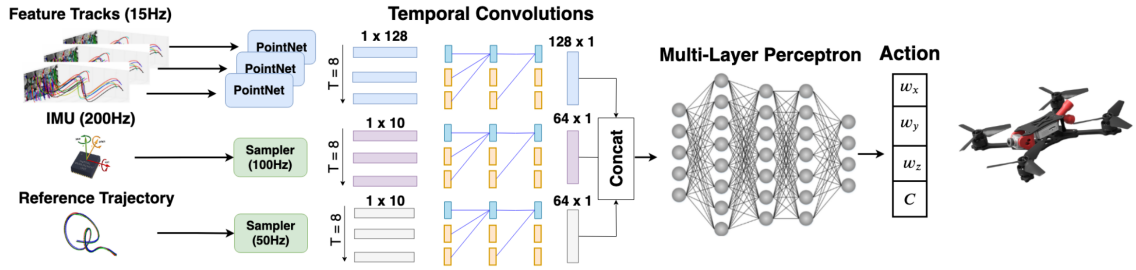


Figure 6.3: Architecture of deep drone aerobatics network[Lit22]

The third and final method is outputting the single thrust for every motor. This method does not require any additional control loop as the network directly controls the motors and correctly represents the true actuation limits of the platform. An example of this method in action is [Lit8], where the authors create a guidance and navigation network that takes as input the state and rpms and outputs the required rpm of each motor for a 3-dimensional quadcopter model taking into account drag, aerodynamic effects and actuator delays.

6.3. Trade-off Control Methods

Having an overview of traditional and learning-based computationally efficient control methods it is now time to perform a trade-off and select the best options to deploy on a Crazyflie for the purpose of drone racing. Once again, the main parameters to keep in mind are:

- The computational efficiency of the algorithm.
- The robustness to model errors.
- The performance it can achieve on the drone.

The algorithms that are traded off are a selection of the previously described methods that might be deployable on the Crazyflie flight controller, they are shown in Table 6.1. In this table one may notice that efficiency has the highest weight, this is due to the fact that the Crazyflie has very limited computational resources, especially on the flight controller. Next in order of importance is the robustness, as nano-drones are more prone to model errors and brushed motors are highly unpredictable so the algorithm needs to be robust. The performance parameter, which measures the extent to which the algorithm effectively utilises a drone's flight envelope, holds significant importance in drone racing. However, measuring this parameter poses a challenge due to variations in the drone models used across different research papers. For this reason, performance is given a lower weight.

ID	High Level Controller	Low Level Controller	Efficiency Weight 3	Robustness Weight 2	Performance Weight 1	Overall
1	Waypoint	Bitcraze PID	9	5	3	37
2	MPC	Bitcraze PID	4	7	6	27
3	Deep Drone Aerobatics[Lit22]	Bitcraze PID	1	6	9	24
4	G&C Net [Lit8]	G&C Net[Lit8]	5	6	8	35

Table 6.1: Trade-off of control methods

Overall, the first approach is the most efficient as there is no planning involved the input of the PID controller is just the required waypoint, the center of the next gate. Without planning, the limitations of the drone are not taken into account and both robustness and performance will suffer from this. The second approach has an MPC as high-level control, thus an optimisation problem has to be solved leading to a high computational load. However, the dynamic model of the drone and its constraints are taken into account, leading to higher robustness and performance. The third method is computationally

expensive due to the temporal convolutions and deep network that need to run sequentially, so even if it leads to a fast drone and robust behaviour, it is not plausible to deploy this on a Crazyflie. Finally, the last approach uses a network for both high and low-level control, this network is quite efficient but needs to run at the same rate as a low-level controller would(500Hz). As for robustness and performance, the score is quite high as this network has access to rpm information, thus it can work with real constraints on the drone. In conclusion, the best approach to deploy on a Crazyflie is the first one due to its simplicity and efficiency. Although, also the fourth one with some network acceleration, see chapter 7, can be tried.

7

Deploying Networks on Edge

Now that the perception and control methods have been determined, the next challenge is deploying them on the Crazyflie. This section focuses on some methods to accelerate and deploy networks on the AI-Deck as this is the major challenge. section 7.1 deals with the deployment pipeline on the AI-Deck, then the next to sections focus on methods of network acceleration namely, quantization (section 7.2) and sparsification(section 7.3). Finally, benchmarks of researchers that already deployed neural networks of the AI-Deck are presented in section 7.4.

7.1. Gapflow

Once a network has been generated and trained it needs to be deployed on the AI-Deck, a board with a GAP8 ultra-low power 9-core processor created by GreenWaves Technologies[Lit13]. This company also developed a pipeline to easily port neural networks on the processor, while simultaneously accelerating the deployment and ensuring high performance and low energy consumption, this tool is called GAPflow, and its pipeline is shown in Figure 7.1.

The process starts with the model in tflite or onnx format, a compressed format for neural networks which saves both architecture and weights. This model is fed to the GAP NNTool, which will map the nodes to the GAP computational node, produce the AutoTiler model, and if necessary quantize the model, see section 7.2.

Next, the AutoTiler Model is fed to the GAP AutoTiler, which will optimise the data movement across the memory hierarchy based on the network and the memory constraints, producing an optimised parallel code with memory movement calls. Finally, this generated code can be dockerized and flashed to the AI-Deck with the help of the crazyradio.

7.2. Quantization

Quantization is a method to shrink and accelerate both training and inference processes of a neural network. With the rise of deep learning and models getting increasingly larger, this method attracted a lot of attention from the scientific community. The goal is to decrease both sizes of models and inference time without affecting performance. To achieve this, real values are converted to 8-bit representations allowing bitwise operations to be used rather than the much slower floating-point operations. This section explains the two main techniques of Quantization, deterministic and stochastic, and the method to apply them to the different components of a neural network. [Lit14]

Deterministic Quantization is when the input values are mapped one-to-one to a quantized value in a predictable and fixed way. Two examples of this technique are rounding and vector quantization.

- **Rounding** is the simplest way to quantize real values. The simplest method was proposed by paper [Lit7], where weights are constrained to only two possible values 1 or -1 following Equation 7.1.

$$x^b = \text{Sign}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{7.1}$$

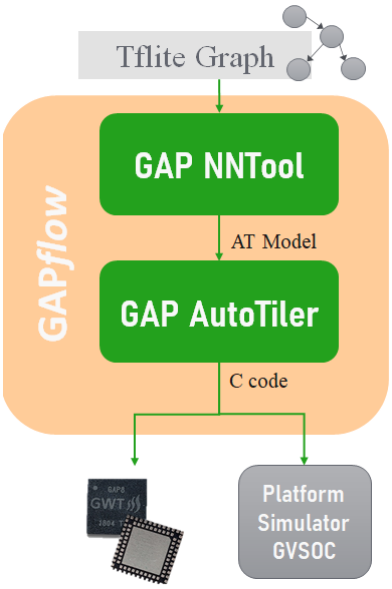


Figure 7.1: Pipeline of gapflow for neural network deployment on the AI-Deck [Lit13]

The method works in forward propagation but error back-propagation will not work and the values will be almost always very close to zero. This led to the development of many other methods. The most notable is proposed by Polino [Lit36], which consists of a general rounding function Equation 7.2, where sc is a scaling function that maps values for a range to 0 and 1 values. And, $\hat{Q}$ is the Quantization function Equation 7.3, which given a Quantization parameter s_q can assign x to the nearest Quantization point.

$$Q(x) = s_q c^{-1}(\hat{Q}(s_q c(x))) \tag{7.2}$$

$$\begin{aligned} \hat{Q}(x, s_q) &= \frac{\lfloor x s_q \rfloor}{s_q} + \frac{\xi}{s} \\ \xi &= \begin{cases} 1 & x s_q - \lfloor x s_q \rfloor > \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{7.3}$$

Rounding is a very simple way to convert real values into quantized ones. However, the network performance may drop drastically. Moreover, rounding decreases the parameter space, making it harder for the training to converge.

- **Vector Quantization** consists in grouping weights and using the centroid of each group to replace the actual weight during inference. On such method [Lit12] performs k-means clustering on the weight matrix. After this, the centroids are fine tuned for better accuracy. This method can significantly reduce memory footprint. In fact, when it was applied to ImageNet it compressed the network 16 to 24 times with an accuracy loss of 1% [Lit12]. The main drawback is that the loss of accuracy caused by k-means cannot be controlled.

Stochastic Quantization is when weights, activations, or gradients are discretely distributed, and the quantized values are sampled from the discrete distribution. The two main techniques are random rounding and probabilistic quantization.

- **Random Rounding** does not have a one-to-one mapping of real and quantized values. The quantized values are sampled from a discrete distribution. A simple example is proposed in [Lit29] and follows Equation 7.4

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{7.4}$$

Where σ is defined as follows in Equation 7.5.

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) \tag{7.5}$$

Therefore, if x is a positive value it will have a high probability of being quantized to 1. Random rounding allows injection of noises into the training process and this has advantages and disadvantages. On the one side, this noise can act as a regulariser and allow conditional computation. On the other hand, it causes oscillations in the loss function during training.

- **Probabilistic Quantization** assumes that weights are distributed according to a probability distribution. Then applies discrete values to weights based on this distribution. For example, in paper [Lit44] the authors developed an Expectation Back-Propagation algorithm to train neural networks, where they first assume some discrete distribution and then update it online during training. The main advantage of this method is that the weights can be forced to have discrete values.

Deterministic quantization is better if the objective of the quantization is to accelerate hardware as it is possible to specify the quantization level in advance and it can be tailored to the hardware. However, stochastic quantization provides more accurate results.

Moving on to the challenges of quantizing different components of a network(weights, activations and gradients). Quantizing weight can reduce the model size and accelerate inference time. However, they make converging harder during the training process, forcing a small learning rate. Quantized weights also make traditional Back-Propagation infeasible, requiring the use of approximation methods.

Quantized activations, inputs/outputs of hidden layers, allow the replacement of inner products with binary operations, speeding up both inference and training. The drawback of this is that there can be a mismatch between the discrete values of the activations and the values computed during the backward propagation step.

Finally, quantized gradients reduce the cost of training of very large models, this is not very applicable in the case of this report as the used model will be small. Nevertheless, this is a very useful feature for back-propagation, but sophisticated algorithms are required to not jeopardise the convergence rate.

The deep learning library TensorFlow, which will be used for this project, has its own implementation of network quantization which focuses on weights and activations. Floating point values are approximated to 8-bit values as using Equation 7.6. Where the r_q is the real value, q_q is the quantized value, Z and S are quantization parameters, zero-point and scale respectively. [Lit19]

$$r_q = M(q_q - Z) \tag{7.6}$$

Weights are represented as int8 two's complement values in the range [-127,127] with the zero-point equal to 0, as they are forced to be symmetric. Activations are also represented as int8 two's complement values but have a range of [-127,128] and a zero-point within the same range.

7.3. Sparsity

Sparsity is a property of neural networks where are a large number of the connections between neurons have zero weights. This phenomenon can occur naturally or be forced upon a network. A great illustration of the effects of sparsity can be seen in Figure 7.2. As sparsity increases in part A, the model experiences reduced noise, leading to improved accuracy and a decrease in size, thereby enhancing performance. Next in part B, the accuracy levels out and performance increased slightly. Finally in part C, the sparsity is very high and the accuracy drops.

The two main questions now are what and when to apply sparsification. Starting from the first question, there are two types of sparsification, model and ephemeral. Model sparsification relates to the fine-grained pruning of weights and neurons of the network. This type shrinks the model and essentially makes a new model still deployable of the same device as the original. Ephemeral sparsification refers to sparsification dependent on the input data applied during the network operations. ReLu and SoftMax operators are an example of this type of sparsification happening naturally. A less common method in this category is the sparsification of gradients which improves performance during training.

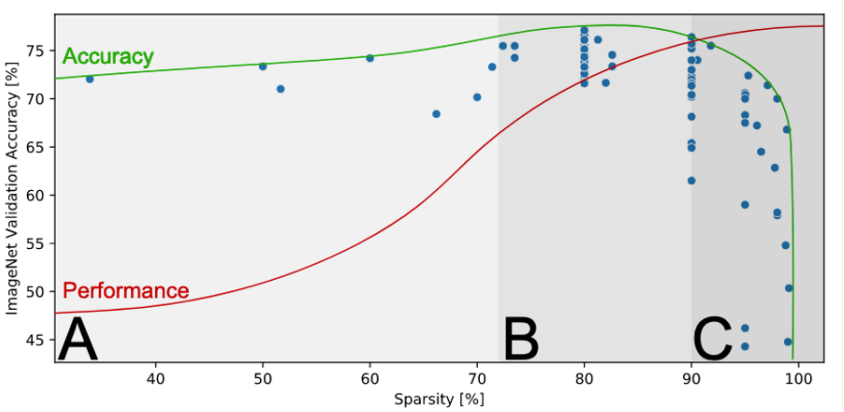


Figure 7.2: Occam's hill showing the test error against the sparsity for ResNet-50 [Lit18] [Lit38]

Moving on to at what point sparsification should be applied. The most trivial option is applying it after training, this usually leads to a significant performance degradation. Moreover, it implies that the full network is trained, which is a costly process. Another method is to schedule sparsification during training. The starting point is a dense model and at every epoch of training, it gets sparsified. This is an extremely flexible method applied in many different ways. For example, the authors of paper [Lit20] propose a combined dense and sparse training in a single schedule. [Lit15] proposes a solution that uses a three step schedule, first dense training, then pruning and retraining, and dense training again. These two methods solve the issue of convergence but the dense models still need to be trained, which comes at a high computational cost. The last category is sparse training, the starting point for the training is an already sparse model, and the training procedure adds and removes weights. Convergence is quite hard to reach with this method[Lit18] .

7.4. Optimal Deployment Strategy

Deploying Computer Vision algorithms on the AI-Deck for real-time on-board image processing is a topic that got plenty of attention recently, many researchers attempted this with different networks with different inference times. Table 7.1 summarises the best performing solutions to this problem. GateNet has been deployed on the AI-Deck, with only Quantization, as a reference.

ID	Title	FPS
1	NanoFlowNet[Lit5]	5.57
2	NanoFlowNet-s[Lit5]	9.34
3	Object Detection CNN[Lit23]	4.30
4	DDND[Lit48]	1.24
5	Tiny-PULP-Dronets[Lit24]	160
6	GateNet	4.55

Table 7.1: Deployment times for different Vision Networks

NanoFlowNet and NanoFlowNet-s are two CNNs from the same paper [Lit5], used for real-time dense optic flow estimation. The network is designed from semantic segmentation networks, and takes as inputs two frames outputting the optic flow field. It is a fully convolutional network with both encoder and decoder. To speed up the inference of the network, the authors of this paper realised that the bottleneck were the convolutional layers. They then solved this problem by applying three different methods. Firstly, they swapped convolutional layers for depth-wise separable convolutions. Secondly,

they relocated the first convolutional layer. Thirdly, images where fed to the network in grayscale reducing by one-third the number of inputs.

The third approach is a single-shot object detection neural network based on MobilenetV2[Lit23]. This network is able to recognise a tin can from a glass bottle and can be expanded for more objects. To decrease latency, the authors employ a width multiplier on the network, reducing the number of parameters and MMAC operations. A width multiplier is a parameter that scales the number of channels or filters in the network.

The fourth approach consists of a fully CNN for depth estimation. The authors of this paper overcome the latency problem by streamlining a network called Lite-Mono, reducing the number of parameters, from $3.1M$ to $310K$. After this, they employ a knowledge distillation framework to train the model. This framework consists of a large teacher model, Lite-Mono in this case, to transfer knowledge to the smaller model, Distilled Depth for Nano Drones(DDND). By doing so, the number of parameters is drastically reduced without a major loss in accuracy.

Finally, in the fifth approach, Tiny-PULP-Dronet, a variant of Dronet, is deployed on the AI-Deck for autonomous navigation. This paper[Lit24] focuses on achieving the highest possible inference speed on the deck. The two main methods employed by the authors are the use of a width multiplier and sparsity.

These different approaches are all effective in accelerating network inference on edge devices, especially on the AI-Deck. A mixture of these approaches will most probably yield the best performance. For Table 7.1, approach 5 is by far the most effective as it achieves 160FPS and should be applied to the network in this project. On the other hand, the first and fourth approaches are more complex to implement. As one requires the employment of knowledge distillation, the other requires substituting convolutional layers.

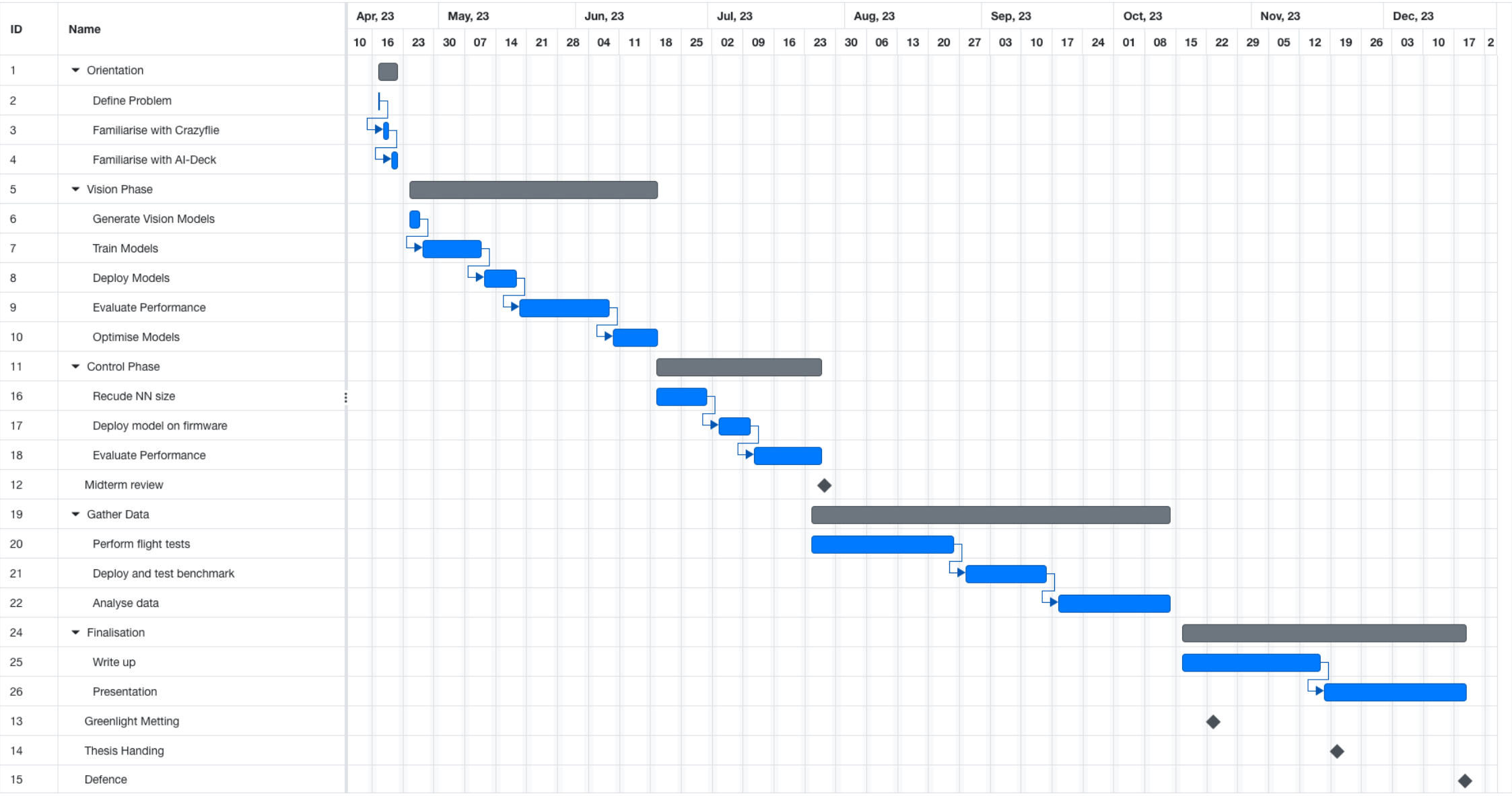
8

Project Plan

In this chapter, the project plan is presented to guide the research in the right direction. The research is divided into three phases namely, vision phase, control phase and gathering data phase. Starting with the vision phase, this part consists in first generating vision neural network or finidng an off the shelf one for the purpose of detecting gates. Next these models will be trained with a sparsification and quantization scheme. Finally, they will be evaluated in terms of accuracy and inference time on both an Intel core and the AI-Deck.

The next phase of the research is the control phase. Here, the focus is shifted to the control and guidance of the drone, so flying through the gates autonomously. The procedure will be to use the optitrack as state estimator, in order to isolate the control algorithm, and try to reduce the size of the G&C Net [Lit8]. If this approach does not work the waypoint approach will be used, see chapter 6.

Finally, the gathering data phase will consist in merging the perception and control, performing test flights and solving any problem that may arise. A Gantt chart presenting the plan with the time window for each phase follows here.


ID
Name
Apr, 23
May, 23
Jun, 23
Jul, 23
Aug, 23
Sep, 23
Oct, 23
Nov, 23
Dec, 23
10 16 23 30 07 14 21 28 04 11 18 25 02 09 16 23 30 06 13 20 27 03 10 17 24 01 08 15 22 29 05 12 19 26 03 10 17 2
1 Orientation
2 Define Problem
3 Familiarise with Crazyflie
4 Familiarise with AI-Deck
5 Vision Phase
6 Generate Vision Models
7 Train Models
8 Deploy Models
9 Evaluate Performance
10 Optimise Models
11 Control Phase
16 Recude NN size
17 Deploy model on firmware
18 Evaluate Performance
12 Midterm review
19 Gather Data
20 Perform flight tests
21 Deploy and test benchmark
22 Analyse data
24 Finalisation
25 Write up
26 Presentation
13 Greenlight Metting
14 Thesis Handing
15 Defence

9

Preliminary Results

This chapter focuses on the preliminary results of research. It starts with an in-depth analysis of the vision networks described chapter 5. This gives insights into the problems related to these networks. section 9.2 proposes a solution to these problems along with the results of its deployment.

9.1. Architecture Analysis

For the best performing networks, DroNet and GateNet/PencilNet(same architecture), an analysis of the inference time of every layer has been carried out using a TensorFlow bench-marking tool, always with an Intel i5 core with a single thread. From Figure 9.1 and Figure 9.2, one can immediately tell that the convolutions are the most computationally expensive layers. In particular, the first and second convolutions, for PencilNet take up around 60% of total time, and the first convolutional layer for DroNet accounts for 40%. This is due to the fact that the image is very large, and the convolution layer has a large number of operations to carry out. In fact, another observation that can be drawn from these graphs is that the rest of the layers do not have a significant impact on the inference time.

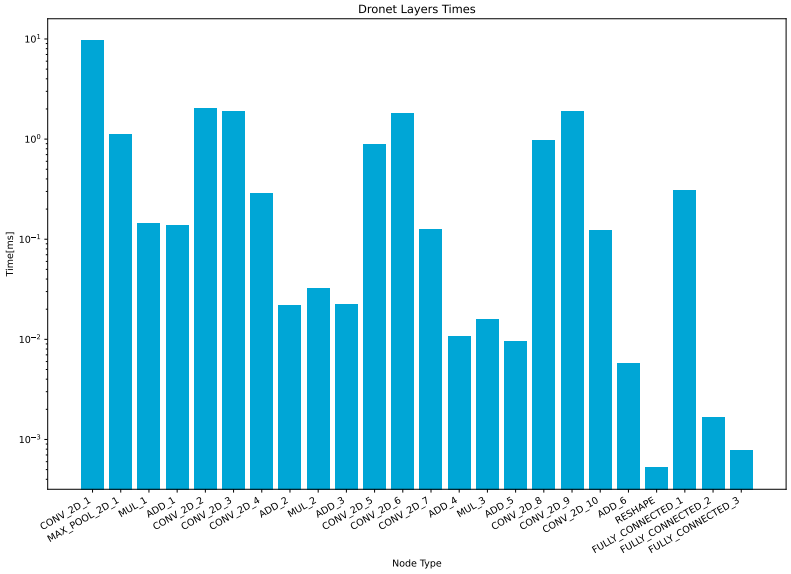


Figure 9.1: Inference time of every layer in Dronet, in the order in which they are run

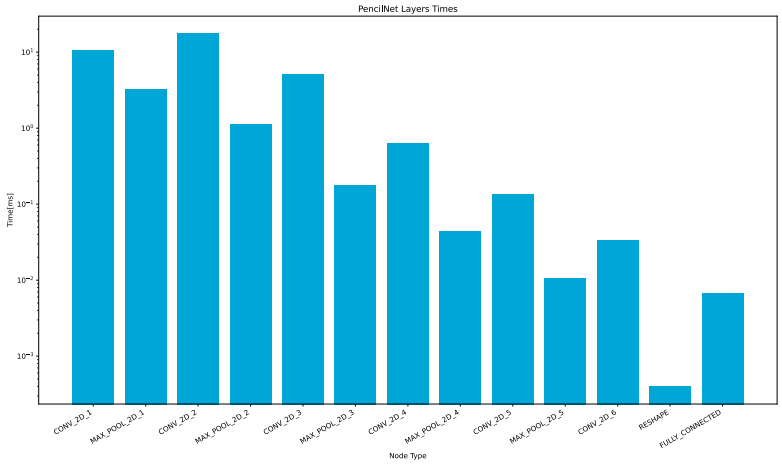


Figure 9.2: Inference time of every layer in PencilNet, in the order in which they are run

9.2. Active Vision

In the previous section, the bottleneck for CNN has been identified as the first convolutional layer. The most trivial solution to this problem is to reduce the number of pixels that need to be processed by this layer. Fewer pixels normally lead to a loss of information, this is where active vision plays a role.

Active vision refers to the use of perception tools to identify the region of interest in an image. In the case of this report, an image is divided into smaller sections and each section is fed to a simple CNN which has 2 outputs namely, if the top left corner is in the section and if not the coordinates of the next section of the image for the network to analyse. Figure 9.3, shows a visualisation of the dataset, where from the centre of every section an arrow to the top left corner of the gate is shown. Once the corner has been identified. All the pixels from right and below the corner are fed to a second CNN which will detect the corners.

Figure 9.3: Active vision dataset visualisation, an arrow from the center of every section points to the top left corner of the gate

A network, with an architecture inspired by GateNet, was trained on these sub-sections. This network has 3 outputs namely, the confidence of if the top left corner was in the image, x and y directions for the unit vector that points to the top left corner. The results can be seen in Figure 9.4, where one can notice that most of the arrows point in the wrong direction and there are two false positive detections of the top left corner. Moreover, from the loss graph, Figure 9.5, is it clear that the model is not learning as

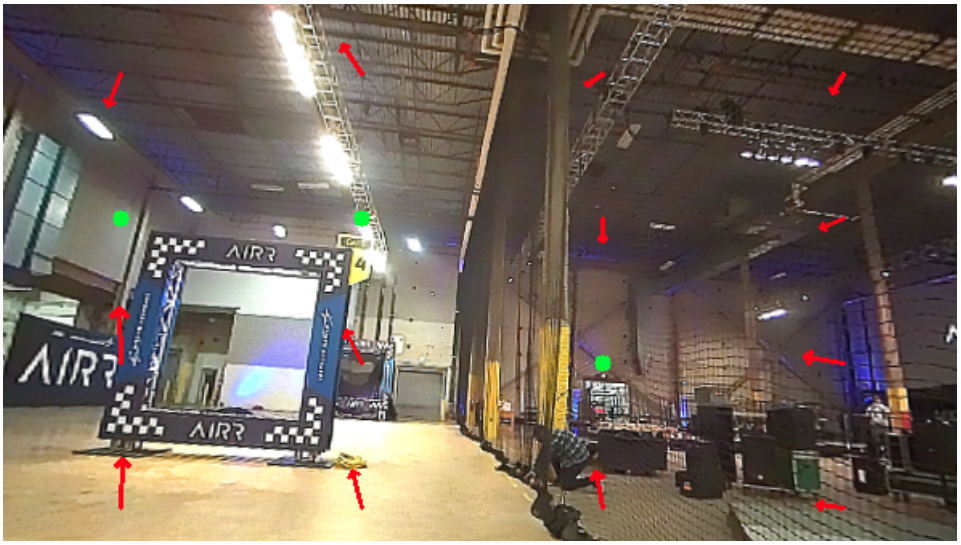


Figure 9.4: Results of the active vision network

the error for the validation never decreases. Running this network on a 36 image dataset, the correct corner was found only 47% of the times and on average the algorithm goes through 7.5 sub-images per image, at a speed of 0.00041092s per sub-image on intel i5 core.

This approach has a low inference time than GateNet or DroNet, but the accuracy is very low. This is due to the fact that it is very hard for a network to understand when in the picture it is looking. For example, looking at the right side of Figure 9.4 it is very difficult to understand if the gate is left or right. Therefore, this approach would need to be trained on a specific dataset with images only of the track that will be flown, resulting in a not-very-robust method.

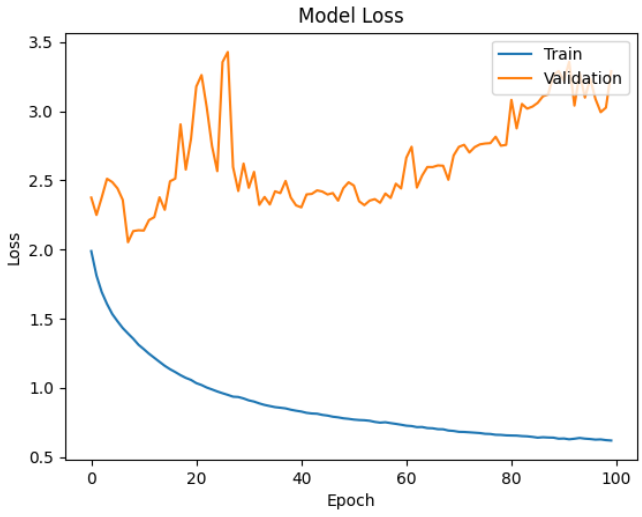


Figure 9.5: Loss of the active vision model

10 Conclusion

In conclusion, the intersection of autonomous drone racing and neural networks offers a promising path for research and development in the field of drone technology. Moreover, it provides an important platform for testing and enhancing the capabilities of autonomous systems, with potential applications in real-world scenarios like search and rescue operations and autonomous inspections.

In the introduction of this literature review, the research questions posed were:

How can state of the art vision and control algorithms for autonomous drone racing be simplified to fit nano-drones, specifically a crazyflie equipped with AI-Deck?

Which perception algorithms perform best in the context of gate detection for autonomous drone racing?

Which control algorithms perform best for high speed autonomous drone racing?

This literature study tried to answer the two sub-questions identifying the state of the art algorithms for perception and control. It did this by dividing the software stack of autonomous nano-drone racing into three distinct parts: perception, control and network acceleration. From the perception side, it was possible to conclude that Neural Network are the most accurate and robust strategy for gate detection. The problem with this strategy is the time it takes to process an image, the bottleneck was identified as the first convolutional layer. A solution was proposed that consists of analysing the image section by section which resulted in a not robust solution. Although, more approaches that try to identify the area of interest should be investigated. The control section provided insight into useful planning and control methods, the most important of which is the G&C Net which from a relative waypoint provides the required motor rpm to reach it. Finally, the network acceleration section introduced two main acceleration methods namely, quantization and sparsification. The former converts the network parameters from floating points to integers allowing integer maths, and boosting speed significantly. The latter, forces a certain percentage of weight to equal zero allowing to easier compression and faster inference.

This literature research lays the groundwork for the subsequent stages of the research project, providing a solid foundation of knowledge and understanding to inform the development and implementation of the autonomous system on a nano-drone. The exploration of cutting-edge strategies and techniques in each stage of the software stack paves the way for advancements in autonomous drone racing and the broaden field of drone technology.

Literature References

[Lit1] Kristoffer Fogh Andersen, Huy Xuan Pham, Halil Ibrahim Ugurlu, and Erdal Kayacan. Event-based navigation for autonomous drone racing with sparse gated recurrent network. 2022.

[Lit2] Suneel Belkhale, Rachel Li, Gregory Kahn, Rowan McAllister, Roberto Calandra, and Sergey Levine. Model-based meta-reinforcement learning for flight with suspended payloads. *IEEE Robotics and Automation Letters*, 6(2):1471–1478, 4 2021. arXiv:2004.11345 [cs].

[Lit3] BitCraze. Crazyflie 2.1, 2020.

[Lit4] K Bousson. 4d trajectory generation and tracking for waypoint-based aerial navigation. 8(3), 2013.

[Lit5] Rik J. Bouwmeester, Federico Paredes-Vallés, and Guido C. H. E. de Croon. Nanoflownet: Real-time dense optical flow on a nano quadcopter. (arXiv:2209.06918), 9 2022. arXiv:2209.06918 [cs].

[Lit6] Maximilian Brunner, Karen Bodie, Mina Kamel, Michael Pantic, Weixuan Zhang, Juan Nieto, and Roland Siegwart. Trajectory tracking nonlinear model predictive control for an overactuated mav. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5342–5348, 2020.

[Lit7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015.

[Lit8] Robin Ferede, Guido de Croon, Christophe De Wagter, and Dario Izzo. An adaptive control strategy for neural network based optimal quadcopter controllers. 2023.

[Lit9] Philipp Foehn, Dario Brescianini, Elia Kaufmann, Titus Cieslewski, Mathias Gehrig, Manasi Muglikar, and Davide Scaramuzza. Alphapilot: autonomous drone racing. *Autonomous Robots*, 46(1):307–320, 1 2022.

[Lit10] Philipp Foehn, Davide Falanga, Nithin Kuppuswamy, Russ Tedrake, and Davide Scaramuzza. Fast trajectory optimization for agile quadrotor maneuvers with a cable-suspended payload. 2017.

[Lit11] Philipp Foehn, Adrià Romero, and Davide Scaramuzza. Time-optimal planning for quadrotor waypoint flight. *Science Robotics*, 6(56), 2021.

[Lit12] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. (arXiv:1412.6115), 12 2014. arXiv:1412.6115 [cs].

[Lit13] GreenWaves. Gapflow, 2021.

[Lit14] Yunhui Guo. A survey on methods and theories of quantized neural networks. (arXiv:1808.04752), 12 2018. arXiv:1808.04752 [cs, stat].

[Lit15] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Shijian Tang, Erich Elsen, Bryan Catanzaro, John Tran, and William J. Dally. DSD: regularizing deep neural networks with dense-sparse-dense training flow. *CoRR*, abs/1607.04381, 2016.

[Lit16] Drew Hanover, Antonio Loquercio, Leonard Bauersfeld, Angel Romero, Robert Penicka, Yunlong Song, Giovanni Cioffi, Elia Kaufmann, and Davide Scaramuzza. Autonomous drone racing: A survey. (arXiv:2301.01755), 1 2023. arXiv:2301.01755 [cs].

[Lit17] Joel Hesch, Dimitrios Kottas, Sean Bowman, and Stergios Roumeliotis. Camera-imu-based localization: Observability analysis and consistency improvement. *International Journal of Robotics Research*, 33:182–201, 01 2014.

[Lit18] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. (arXiv:2102.00554), 1 2021. arXiv:2102.00554 [cs].

[Lit19] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. (arXiv:1712.05877), 12 2017. arXiv:1712.05877 [cs, stat].

[Lit20] Xiaojie Jin, Xiaotong Yuan, Jiashi Feng, and Shuicheng Yan. Training skinny deep neural networks with iterative hard thresholding methods. *CoRR*, abs/1607.05423, 2016.

[Lit21] Elia Kaufmann, Mathias Gehrig, Philipp Foehn, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Beauty and the beast: Optimal methods meet learning for drone racing. (arXiv:1810.06224), 3 2019. arXiv:1810.06224 [cs].

[Lit22] Elia Kaufmann, Antonio Loquercio, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Deep drone acrobatics. (arXiv:2006.05768), 6 2020. arXiv:2006.05768 [cs].

[Lit23] Lorenzo Lamberti, Luca Bompani, Victor Javier Kartsch, Manuele Rusci, Daniele Palossi, and Luca Benini. Bio-inspired autonomous exploration policies with cnn-based object detection on nano-drones. (arXiv:2301.12175), 2 2023. arXiv:2301.12175 [cs, eess].

[Lit24] Lorenzo Lamberti, Vlad Niculescu, Michał Barciś, Lorenzo Bellone, Enrico Natalizio, Luca Benini, and Daniele Palossi. Tiny-pulp-dronets: Squeezing neural networks for faster and lighter inference on multi-tasking autonomous nano-drones. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, page 287–290, 6 2022.

[Lit25] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual–inertial odometry using nonlinear optimization. *Int. J. Rob. Res.*, 34(3):314–334, 3 2015.

[Lit26] Shuo Li, Erik Horst, Philipp Duernay, Christophe De Wagter, and Guido C. H. E. Croon. Visual model-predictive localization for computationally efficient autonomous racing of a 72-g drone. *Journal of Field Robotics*, 37(4):667–692, 6 2020.

[Lit27] Shuo Li, Michaël M. O. I. Ozo, Christophe De Wagter, and Guido C. H. E. de Croon. Autonomous drone race: A computationally efficient vision-based navigation and control strategy. *CoRR*, abs/1809.05958, 2018.

[Lit28] Shuo Li, Michaël M.O.I. Ozo, Christophe De Wagter, and Guido C.H.E. de Croon. Autonomous drone race: A computationally efficient vision-based navigation and control strategy. *Robotics and Autonomous Systems*, 133, 11 2020.

[Lit29] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. (arXiv:1510.03009), 2 2016. arXiv:1510.03009 [cs].

[Lit30] Antonio Loquercio, Ana I. Maqueda, Carlos R. del Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 4 2018.

[Lit31] Matlab. plannerrrt, 2019.

[Lit32] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. pages 2520–2525, 2011.

[Lit33] Anastasios I. Mourikis and Stergios I. Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *2007 IEEE International Conference on Robotics and Automation, ICRA'07*, Proceedings - IEEE International Conference on Robotics and Automation, pages 3565–3572, 2007. Copyright: Copyright 2011 Elsevier B.V., All rights reserved.; 2007 IEEE International Conference on Robotics and Automation, ICRA'07 ; Conference date: 10-04-2007 Through 14-04-2007.

[Lit34] H. X. Pham, I. Bozcan, A. Sarabakha, S. Haddadin, and E. Kayacan. Gatenet: An efficient deep neural network architecture for gate perception using fish-eye camera in autonomous drone racing. pages 4176–4183, 2021.

[Lit35] Huy Xuan Pham, Andriy Sarabakha, Mykola Odnoshyvkin, and Erdal Kayacan. Pencilnet: Zero-shot sim-to-real transfer learning for robust gate perception in autonomous drone racing. (arXiv:2207.14131), 7 2022. arXiv:2207.14131 [cs].

[Lit36] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. (arXiv:1802.05668), 2 2018. arXiv:1802.05668 [cs].

[Lit37] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[Lit38] Carl Rasmussen and Zoubin Ghahramani. Occam' s razor. In *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000.

[Lit39] Leticia Oyuki Rojas-Perez and Jose Martinez-Carranza. Deeppilot: A cnn for autonomous drone racing. *Sensors*, 20(16):4524, 2020.

[Lit40] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. 2015.

[Lit41] Davide Scaramuzza and Zichao Zhang. Visual-inertial odometry of aerial robots. 2019.

[Lit42] Jingnan Shi. Perspective-n-point: P3p, 2022.

[Lit43] Ewoud Smeur, Q.P. Chu, and Guido de Croon. Adaptive incremental nonlinear dynamic inversion for attitude control of micro air vehicles. *Journal of Guidance, Control, and Dynamics*, 39:1–12, 12 2015.

[Lit44] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[Lit45] Zachary Teed, Lahav Lipson, and Jia Deng. Deep patch visual odometry. 2023.

[Lit46] Christophe De Wagter, Federico Paredes-Vallés, Nilay Sheth, and Guido de Croon. The artificial intelligence behind the winning entry to the 2019 AI robotic racing competition. *CoRR*, abs/2109.14985, 2021.

[Lit47] Daniel J. Webb and Jur van den Berg. Kinodynamic rrt*: Asymptotically optimal motion planning for robots with linear dynamics. pages 5054–5061, 2020.

[Lit48] Ning Zhang, Francesco Nex, George Vosselman, and Norman Kerle. Channel-aware distillation transformer for depth estimation on nano drones. (arXiv:2303.10386), 3 2023. arXiv:2303.10386 [cs].

Bibliography

[1] Maximilian Brunner, Karen Bodie, Mina Kamel, Michael Pantic, Weixuan Zhang, Juan Nieto, and Roland Siegwart. Trajectory tracking nonlinear model predictive control for an overactuated mav. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5342–5348, 2020.

[2] Christophe De Wagter, Federico Paredes-Vallé, Nilay Sheth, and Guido de Croon. The sensing, state-estimation, and control behind the winning entry to the 2019 artificial intelligence robotic racing competition. *Field Robotics*, 2(1):1263–1290, March 2022.

[3] Robin Ferede, Christophe De Wagter, Guido de Croon, and Dario Izzo. End-to-end reinforcement learning for time-optimal quadcopter flight. In *ICRA 2024*, 2023.

[4] Yunhui Guo. A survey on methods and theories of quantized neural networks. (arXiv:1808.04752), 12 2018. arXiv:1808.04752 [cs, stat].

[5] Drew Hanover, Antonio Loquercio, Leonard Bauersfeld, Angel Romero, Robert Penicka, Yunlong Song, Giovanni Cioffi, Elia Kaufmann, and Davide Scaramuzza. Autonomous drone racing: A survey. (arXiv:2301.01755), 1 2023. arXiv:2301.01755 [cs].

[6] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(79767976):982–987, August 2023.

[7] Antonio Loquercio, Ana I. Maqueda, Carlos R. del Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 4 2018.

[8] H. X. Pham, I. Bozcan, A. Sarabakha, S. Haddadin, and E. Kayacan. Gatenet: An efficient deep neural network architecture for gate perception using fish-eye camera in autonomous drone racing. pages 4176–4183, 2021.

[9] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[10] Jingnan Shi. Perspective-n-point: P3p, 2022.

[11] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019.