NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

POSTGRADUATE PROGRAM
ADVANCED INFORMATION SYSTEMS

MSc THESIS

Autonomous Drones for Trail Navigation using DNNs

Georgios A. Kalampokis

Supervisor: **Miltiadis Kyriakakos,** Laboratory Teaching Staff

ATHENS

JANUARY 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΡΟΗΓΜΕΝΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αυτόνομη Πλοήγηση μη Επανδρωμένων Αεροσκαφών σε Δασικά Μονοπάτια με χρήση τεχνικών Βαθιάς Μάθησης

Γεώργιος Α. Καλαμπόκης

Επιβλέπων: **Μιλτιάδης Κυριακάκος,** Ε.ΔΙ.Π.

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2022

MSc THESIS

Autonomous Drones for Trail Navigation using DNNs

Georgios A. Kalampokis
S.N.: M1300

January 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αυτόνομη Πλοήγηση μη Επανδρωμένων Αεροσκαφών σε Δασικά Μονοπάτια με χρήση τεχνικών Βαθιάς Μάθησης

Γεώργιος Α. Καλαμπόκης
Α.Μ.: Μ1300

Ιανουάριος 2022

ABSTRACT

Autonomous Navigation is a technology that had been developing significantly since the early 2010’s. It gives the ability to a vehicle to plan its path and execute its plan without human interaction. Some of the fields that Unmanned Vehicles may apply in are autonomous driving, surveillance, security monitoring, or crisis management and risk assessment activities. Despite the wide acceptance in the field of automotive industry, there is a limited development of applications for aerial vehicles such as drones, due to expensive hardware and complex software synthesis.

This thesis proposes the design and implementation of a prototype drone stack that is able to autonomously navigate through a forest trail path without having prior knowledge of the surrounding area. It uses a 3 level vision system: (i) a deep neural network (DNN) for estimating the view orientation and lateral offset of the vehicle with respect to the trail center, (ii) a DNN for object detection and (iii) a Guidance system for obstacle avoidance.

Our drone stack is consisted of a DJI Matrice 100 drone integrated with sensors such as camera, IMU, GPS, collision avoidance system, and is retrofitted with a Jetson TX2 supercomputer module that runs all computer vision and decision making tasks in real time. We provide details on software stack used, as also for implementation. For training of the trail path DNN, we used the IDSIA Swiss Alps trail dataset along with our custom dataset that we created from footage within the campus.

SUBJECT AREA: Machine Learning, Robotics

KEYWORDS: autonomous navigation, neural networks, DJI Matrice 100, Jetson TX2, ROS

ΠΕΡΙΛΗΨΗ

Η αυτόνομη πλοήγηση είναι μια τεχνολογία που έχει σημειώσει σημαντική ανάπτυξη από τις αρχές της δεκαετίας του 2010. Δίνει την δυνατότητα σε ένα όχημα να σχεδιάσει την δρομολόγησή του και να εκτελέσει το σχέδιο δρομολόγησης δίχως ανθρώπινη παρέμβαση. Κάποια από τα πεδία ειδίκευσης όπου τα Μη Επανδρωμένα Αεροσκάφη έχουν εφαρμογή είναι: η αυτόνομη πλοήγηση αυτοκινήτων, συστήματα παρακολούθησης, συστήματα ασφαλείας, συστήματα διαχείρισης κρίσεων και εκτίμησης κινδύνου. Παρόλη την ευρεία αποδοχή και ανάπτυξη στον χώρο της αυτοκινητοβιομηχανίας, η ανάπτυξη εφαρμογών για μη επανδρωμένα εναέρια μέσα (τύπου drone) είναι περιορισμένη, λόγω του ακριβού εξοπλισμού και των σύνθετων λογισμικών που χρησιμοποιούν.

Στην παρούσα διπλωματική εργασία, προτείνεται ο σχεδιασμός και η υλοποίηση ενός πρότυπου drone που έχει τη δυνατότητα αυτόνομης πλοήγησης σε δασικό μονοπάτι χωρίς πρότερη γνώση του περιβάλλοντα χώρου. Χρησιμοποιεί σύστημα τεχνητής όρασης τριών επιπέδων: (i) ένα νευρωνικό δίκτυο βάθους (DNN) για εκτίμηση πλευρικής μετατόπισης και προσανατολισμού ως προς το κέντρο του μονοπατιού, (ii) ένα DNN για αναγνώριση αντικειμένων, και (iii) ένα σύστημα αποφυγής εμποδίων.

Η πρότυπη κατασκευή μας αποτελείται από την drone πλατφόρμα Matrice 100 της DJI, εξοπλισμένη με αισθητήρες όπως μια κάμερα υψηλής ανάλυσης, IMU, GPS, και σύστημα αποφυγής εμποδίων. Επιπροσθέτως φέρει την πλακέτα υπέρ-υπολογιστή Jetson TX2 της NVIDIA όπου τρέχουν όλες οι εργασίες τεχνητής όρασης και οι αλγόριθμοι λήψης αποφάσεων σε πραγματικό χρόνο. Τεχνικές λεπτομέρειες παρέχονται για το υλικό και λογισμικό που χρησιμοποιήθηκε. Για την εκπαίδευση του DNN εύρεσης μονοπατιού, χρησιμοποιήσαμε ένα σύνολο δεδομένων από τις Ελβετικές Άλπεις διαθέσιμο από το ερευνητικό κέντρο IDSIA, σε συνδυασμό με ένα δικό μας που δημιουργήσαμε με καταγραφές από δασικά μονοπάτια εντός της Πανεπιστημιούπολης Ιλισίων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανική Μάθηση, Ρομποτική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: αυτόνομη πλοήγηση, νευρωνικά δίκτυα, DJI Matrice 100, Jetson TX2, ROS

To my Family

ACKNOWLEDGEMENTS

I would like to thank my supervisor and scientific coordinator of this diploma thesis Dr. Miltiadis Kyriakakos, who gave me the opportunity to work on the field of machine learning and robotics. With his constant support, we finally managed to successfully complete our initial idea in developing an autonomous Micro Aerial Vehicle (MAV).

Also, I would like to thank Marina for her patience and support all these years. Special thanks to my family for their continuous support during my studies.

CONTENTS

LIST OF FIGURES

LIST OF IMAGES

LIST OF TABLES

PREFACE

This project has been developed since February 2021 in the University of Athens at the department of Informatics and Telecommunications as my postgraduate thesis. Hardware used was sponsored by the Pervasive Computing Research Group (p-comp) of the same department.

Athens, September 2021

1. INTRODUCTION

In recent years, autonomous navigation applications for UAVs have grown significantly and are used for industrial, military, civilian and research purposes. Some of the fields that these applications for UAVs are been applied are: autonomous driving, surveillance, security monitoring, mapping, crisis management, search and rescue activities, risk assessment and recreational activities such as personal video shooting. In comparison to Ground Vehicles, UAVs stands out for their advantage in flying around unstructured outdoor environments giving them the ability to function at higher speeds and variable heights, overcome non-traversable ground obstacles, and covering distances in a straight line.

Autonomous flight in unstructured environments such as forests is a challenging task that is still under research and development. A solution to this problem could be helpful for many applications like wilderness mapping, search and rescue activities, or even for fire monitoring during high risk seasons. The most efficient and safest way to explore a forested area is by following a trail path (such as hikers and/or moto crossers are using). The most suitable vehicle for this task is a MAV flying under the tree canopy, which, due to its size, is more collision resilient. It can cover long distances using optical sensors with minimal risk.

Trail following is a complex task. MAV needs to recognize the trail, and make the appropriate decisions in order to keep its trajectory close to the centre line. Extracting trail figure from a monocular image is an extremely difficult pattern recognition problem, sometimes even for humans. Its appearance (shape and width) may vary in the passing of seasons; often seamlessly blend with the surrounding area, leading to ambiguous boundaries.

Another problem in autonomous navigation is obstacle avoidance. Low-flying MAVs need to be environmentally aware from colliding with branches, hikers, animals or any other objects using the trail. This can be dealt as an object detection problem or as a depth estimation problem or as a combination of these two.

In this project: (a) we introduce a hardware/software synthesis of a MAV for autonomous navigation in a forested trail path. It uses a DNN architecture specialized in trail detection estimating both view orientation and lateral offset. This is the very first system of which we are aware of that combines a DJI M100 drone platform with a Jetson TX2 module. (b) We created a dataset from footage within the University campus for retraining the DNN model to recognize the lateral offset on the trail. This also makes the model more adaptive to local vegetation characteristics. (c) For object detection service, we did a comparison between well-known algorithms and evaluated them in terms of accuracy and efficiency.

The rest of the thesis is as follows: In Chapter 2 the problem formulation and the rationale behind the algorithms used are introduced. The implementation of the algorithms, the hardware and software setup, the training and the experiment results are provided in Chapter 3. The conclusions and future work are given in Chapter 4.

1.1 Previous Work

In several previous works, finding trail paths from a single image is treated as a semantic segmentation problem. The goal is to extract, group and outline these visual features that define a trail path. Rasmussen et al. [3] define these features using appearance contrast while Santana et al. [4] relies on image conspicuity. Both approaches can be viewed as a problem of saliency estimation. Saliency quantifies each pixel according to its unique colour quality in a colour image. For example a group of pixels that represent a specific coloured object on a uniform background will be denoted with higher saliency than the pixels on the background. So in case of an image that contains a trail path with visual differences to its surrounding areas, saliency estimation will output high values for trail pixels and low values for everywhere else. In Levin and Weiss [5] salience data are aggregated, using also deep learning techniques by considering both top-down and bottom-up cues simultaneously, in order to extract segments of the trail and other obstacles in the image. Semantic segmentation approach was also followed in another project of Rasmussen et al. [7] that used stereo cameras along with a laser range finder device in order to navigate a wheeled robot through a trail path.

Another approach for the trail perception problem is to transform it to an image classification problem. Giusti et al. [1] proposed a method of predicting the view orientation of the MAV compared to the trail path direction. They developed a DNN that was trained with footage from a head-mounted rig consisted by three cameras (each one aiming at left/straight/right of the trail path’s center line). Then, the trained DNN was outputting steering commands for keeping MAV close to the center line. This project was based upon previous work of Rasmussen et al. [7] that used a ground vehicle. Smolyanskiy et al. [2] extended this, by enriching the method to travel along trail path’s center line by computing the lateral offset from the center of the trail path. They were inspired by M. Bojarski et al. [11], an NVIDIA’s DNN-controlled self-driving car that was using three different direction cameras on-board. Our project is actually based on DNN architectures used in [1] and [2], trained with the IDSIA forest trail dataset [6] and our custom dataset that was created from three cameras on a wide baseline rig. This makes our system capable of estimating both view orientation and lateral offset within the trail.

Zhilenkov and Epifantsev [9] proposed a system with autonomous environment recognition and decision making for forest trail navigation, equipped with three different direction cameras, assisted by pre-image processing. Maciel-Pearson et al. [8] proposed a similar method, in which input images were horizontally split into three equal parts, and the probability of the existence of the trail is computed for each part, concluding to a heading direction. Palossi et al. [10] proposed a neural network for distinguishing and follow the running tracks in the image, for use in a low powered on-board computer.

Several studies introduce visual-based processing methods for dealing with obstacle avoidance. Alvarez et al. [12] uses a structure-from-motion (SfM) algorithm to infer depth from a single front camera image. Bry et al. [13] uses a combination of two sensors, an IMU with a laser range finder, for localization of the MAV flying in indoor environment. Fraundorfer et al. [14] use a combination of front facing stereo-cameras with a downward facing single camera for map building and state estimation. Scaramuzza et al. [15] use an IMU with three cameras.

Obstacle avoidance problem is also resolved through feature matching algorithms. Mori et al. [16] proposes the SURF (Speeded-Up Robust Features) algorithm and Al-Kaff et al. [17] the SIFT (Scale-Invariant Feature Transform). The only drawback of these two algorithms is time complexity which is proportional to input image resolution and total number of key points. As shown in Drews et al. [18], in complex environments, such as forests, more key points are generated that lead to the need of more computational power, which makes these algorithms restrictive in running at real-time on such on-board computer modules. They are also unsuitable on detecting features in case of moving obstacles [17].

Smolyanskiy et al. [19] introduce a novel semi-supervised learning algorithm by training a deep stereo neural network for depth estimation. They also created a minimized version of the same stereo DNN, allowing it to run on an embedded GPU such as Jetson TX2.

2. RATIONALE AND PROBLEM FORMULATION

For an MAV, to succeed in autonomous navigation through unseen unstructured outdoor environments, it has to develop a logic that, assisted by the sense of sight, will keep it focused on its goal. This logic can be expressed through a group of pre-trained neural networks. In this chapter we present the strategy followed on how to reach the primary goal, by simplifying the use of each neural network separately.

2.1 Definition of the problem

Consider an image that contains a segment of a trail path somewhere in a forest or a mountain, and we need to perceive that trail from that image. We can assume that this image is acquired from a single monocular camera from the view-point of a hiker that has an average height of 1.7 meters. It is a reasonable choice because in most cases it is a height free of obstacles, providing a promising overview of the surrounding area.

As described in [1], we adopt the same method by considering the problem as classification rather than regression. By processing the image as a whole: (i) make data acquisition and labelling a simple task, (ii) model is less prone to noise, (iii) there is no need in defining separate characteristic features of trails, which is a computational needy task given the variability of their appearance.

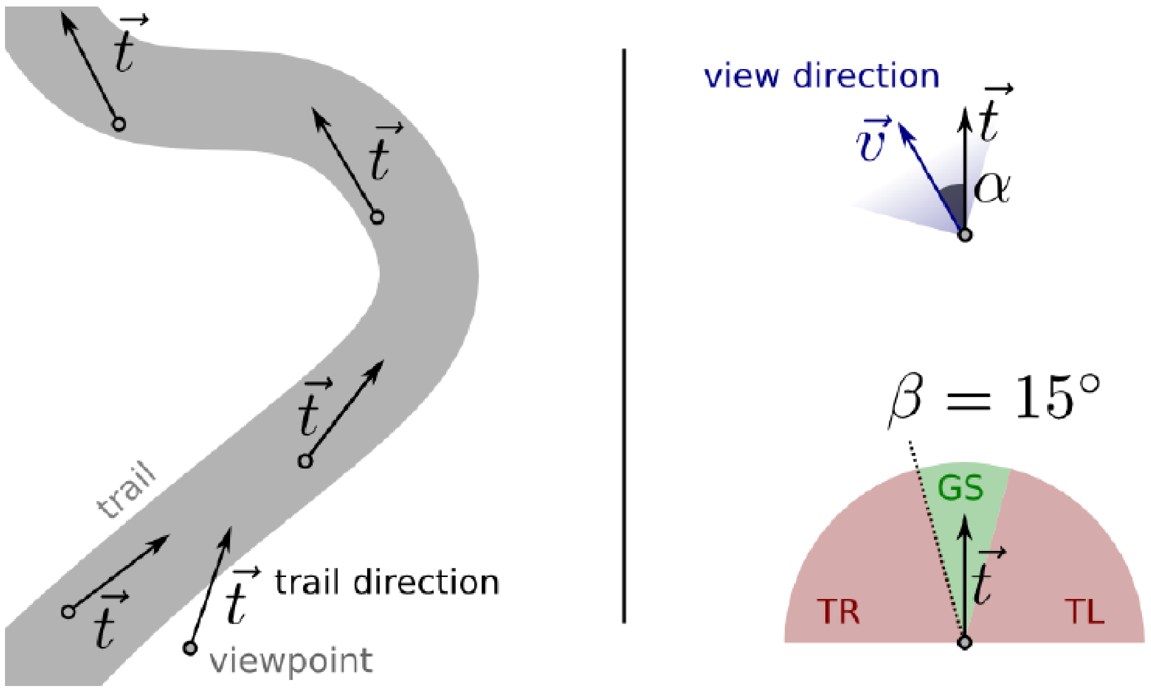


Figure 1: Left: $\vec{t}$ is the trail's center line. Right: $\vec{v}$ is the camera's view point, a,b are angles (see text for details) [1]

In the left pane of Figure 1, we present the trail's center line; $\vec{t}$ is the horizontal direction that a walker should follow in order to traverse the trail path, remaining as close to the center of the trail. In the right pane, we define $\vec{v}$ as the direction of the camera's view point, and $\vec{a}$ is the signed angle between $\vec{v}$ and $\vec{t}$.

According to the angle $\vec{a}$ between the view-point of the camera and the trail's center line direction, three classes are defined, each one corresponding to a steering action for the carrier of the camera that needs to follow in order to remain closest to the center of the trail. So, the three classes that define the heading directions are:

- Turn Left (TL): if $-90^0 < a < -\beta$ then trail direction is on the left of the image.
- Go Straight (GS): if $-\beta \leq a < +\beta$ then trail direction is on (or close to) the center of the image.
- Turn Right (TR): if $+\beta \leq a < +90^0$ then trail direction is on the right of the image.

For an input image, the goal is to classify it in one of the pre mentioned classes. Angle β is set to 15^0. For the special case that the absolute value of angle $\vec{a}$ exceeds 90^0, we consider that image does not contain a trail path, so the inferred class is not Go Straight.

In order to increase the performance of trail following by making the UAV to converge to the center of the path, we used the modified network as described in [2] that introduces an additional three classes as output for lateral offset (shifted left / center / shifted right) compared to trail centre. Without these classes, the MAV may fly in parallel to the trail's central line but close to the edge, causing collisions with tree branches or other obstacles. DNN will correct this orientation error only if it has the information of the lateral offset.

2.2 Trail Following

The DNN used in the current project for trail following, determines the head direction (HD) and recognizes the lateral offset (LO) position of the MAV with respect to the trail path. It consists of a modified version of the standard ResNet-18 [24] resulting to a double headed fully connected output layer. The overall network architecture is given in Figure 2. It consists of successive pairs of convolutional and max-pooling layers, followed by several fully connected layers. As outputs it gives three classes for the view orientation of the trail and three classes for the lateral offset of the trail. The latest three classes are essential to accurate state estimation increasing reliability to trail following.

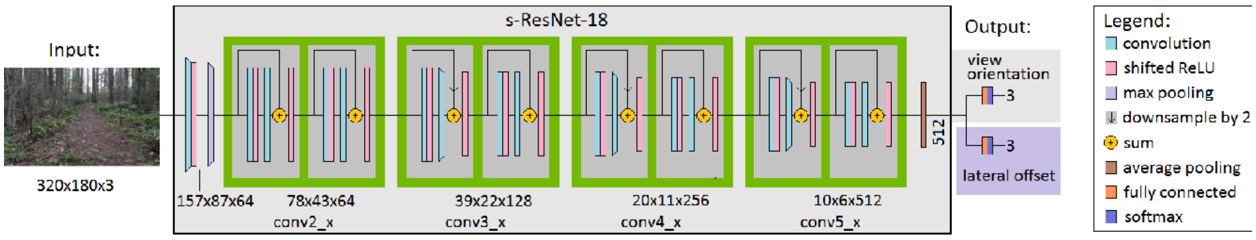


Figure 2: Trail following DNN architecture [2]

The input image is resized to a size of $3 \times 320 \times 180$[1] pixels, which are fed directly to the neurons of the input layer. The DNN outputs probabilities of the input image for the classes TL, GS, TR, and predictions of the lateral offset of left, centre and right.

[1] The image size is 320 x 180 and comes in three chrominance of red, green and blue.

2.2.1 Dataset Acquisition and Pre-processing

In order to train our model, we need to collect at first a satisfying amount of representative data. The appearance variability of the trail paths and their surrounding areas leads to a number of factors that need to be taken into consideration such as: the vegetation types, the local topography, lighting conditions, etc. So, our dataset is consisted of many different long distance trails, with different times in the day (early morning/noon/late afternoon) and weather conditions, covering different locations with varying vegetation during different seasons (summer/autumn/spring).

Dataset acquisition setup is given in Figure 3. For the head direction training, we used the IDSIA forest trail dataset [6]. The footage is from a three head-mounted cameras, (aiming left 30^0, straight and right 30^0), with the FOV (field of view) of each camera partially overlapping, and all three cover in total a 180^0 view. It covers 7 Km of trails on attitudes ranging between 300 m and 1200 m, resulting to an 8 hours Full HD video at 30 fps. As shown in the left pane of Figure 3, a corresponding label is given to each camera according to the direction it shows. An image that shows the left side of the trail is labeled as “turn right” and vice-versa. The trained model sorts input images in three classes: turn left (TL), go straight (GS), and turn right (TR). These are commands for trail following, maintaining the MAV between borderlines.

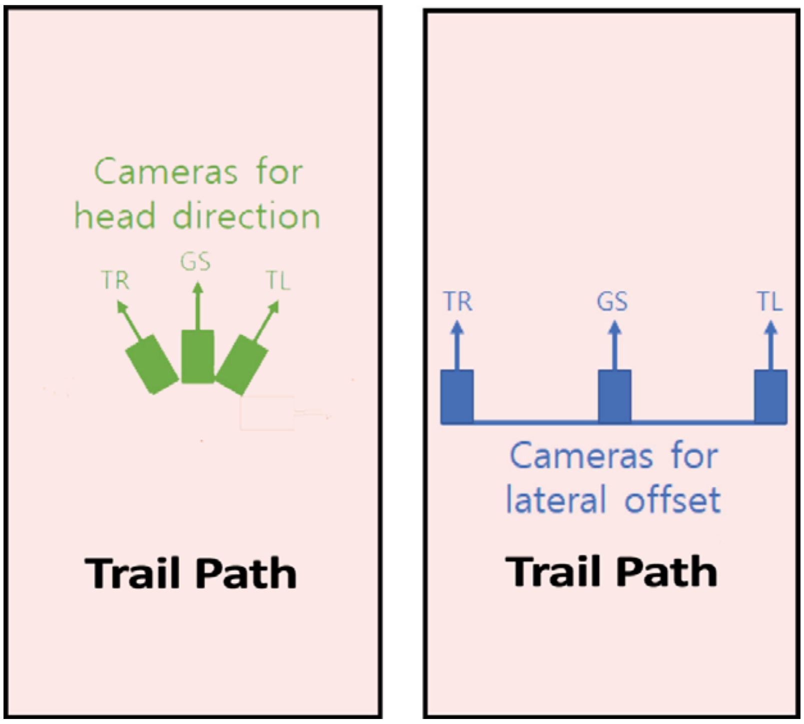


Figure 3: Data acquisition for: Left: head direction commands Right: lateral offset commands

For training of the lateral offset layers of our network, we created our own dataset from footage within the Ilisia University Campus and the surrounding area of Hymettus mountain. As shown in the right pane of Figure 3, we used three Full HD cameras mounted on a 1 m baseline rig, with distance between adjacent cameras set to 0.5 m. The footage recorded resolution was Full HD (1920×1080) at 30 fps with FOV $120^0 \times 90^0$. Across the entire route, a hiker was holding the rig in front of his chest, walking to the center of the path. Because of the fisheye distortion, all videos where initially cropped to horizontal 60^0 FOV, and then used for training the 3 class lateral offset layers. A detailed reference on data pre-processing can be found in Appendix II. So, in addition to head direction estimation, the trained model also predicts the lateral position offset by making a second sort of the input images into categories turn left (TL), go straight (GS), and turn right (TR). These commands help the MAV to get closer to the centre of the path.

2.2.2 Training the Network

The IDSIA trail dataset contains a total of 24.474 frames. A split in training and testing sets was done similar to [1] containing 17.119 and 7.355 frames respectively. In each set, images from each class are equally distributed. Our dataset contains a total of approximately 5.000 images. From these, 4.000 images with their labels were used for training and the rest 1.000 were used for testing.

In the stage of pre-process, data augmentation is done through a set of modifications. Horizontal mirroring is applied randomly in some images followed by label changes respectively. Also other modifications applied are: random contrast with 0.2 radius, random brightness with 0.2 radius, sharpness with 0.3 radius, saturation with 0.4 radius, random crops, and random affine distortions with $\pm 10\%$ scaling, $\pm 10\%$ translation, $\pm 15^0$ rotations.

In the first step, we train the head direction of the network (hyper-parameters in Table 1) through the UI of the NVIDIA DIGITS framework (see Appendix II.D). We use only the IDSIA dataset and the DNN is trained with backpropagation for 20 epochs, with a batch size of 64, and a base learning rate set to 0.001. In addition, a loss function is used (as proposed in [2]) to prevent overconfidence in the network. In the second step, we add lateral offset ability by re-training the network exported from the previous step with same parameters. This time, we use our custom dataset from the University campus.

Table 1: Hyper-Parameters for training the trail following DNN

Parameter	**Value**
Training Epochs (Iterations)	20
Batch Size	64
Base Learning Rate	0.001
Solver Type	Nesterov's accelerated gradient (NAG)
Policy	Polynomial Decay
Loss Function	Cross Entropy (CE) + Entropy Reward (ER) + Side Swap Penalty (SSP)

A network (model) that fits exactly against to its training data is called «overfitted». The drawback is that it memorizes the noise of the training data and it doesn't generalize well to new data. In our case, the MAV results in delayed turning left or right, due to high noise in «Go straight» class. To make our model less confident, we use the loss function [2]:

$$L = -\sum_{i} p_i \cdot \ln(y_i) - \lambda_1 \cdot \left(-\sum_{i} y_i \cdot \ln(y_i)\right) + \lambda_2 \cdot \varphi(y)$$

It is consisted of three terms: Cross Entropy (CE), Entropy Reward (ER), and Side Swap Penalty (SSP). It is used in both head direction and lateral offset training.

In Table 2, the parameters that the loss function contains are described. The parameters that reduce network confidence are smoothed labels used at first term and the entropy reward term.

Table 2: Loss Function Parameters

Loss Function Parameter	**Description**
$\mathbf{p_i}$	Smoothed ground truth label
$\mathbf{y_i}$	Category prediction $i \in \{left, center, right\}$
$\mathbf{y}$	$[y_i, \lambda_1, \lambda_2]$
$\boldsymbol{\varphi}(\mathbf{y})$	$\begin{cases} y_{\text{left}}, \text{if } \hat{\imath} = \text{right} \\ y_{\text{right}}, \text{if } \hat{\imath} = \text{left} \text{ with } \underbrace{\hat{\imath} = \text{argmax}_i p_i}_{\text{ground truth category}} \\ 0, \text{if } \hat{\imath} = \text{center} \end{cases}$

2.2.3 Steering Command Controller (Waypoint Computation)

The trail following DNN takes as input a monocular camera image, and returns two arrays of three items each, representing category predictions for head direction y_i^{HD} and lateral offset y_i^{LO}. Predictions are expressed as softmax values. The standard softmax function is used: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \text{for } i - 1, \ldots, K \text{ and } z = (z_1, \ldots, z_K) \in R^K$. In general, it means that for any input, the outputs must be all positive and they must sum to unity.

Predictions of Head Direction (HD)	Predictions of Lateral Offset (LO)
$\left.\begin{matrix} y_{\text{left}}^{\text{HD}} \\ y_{\text{straight}}^{\text{HD}} \\ y_{\text{right}}^{\text{HD}} \end{matrix}\right\} \overset{sum}{\Longrightarrow} \sum y_i^{\text{HD}} = 1$	$\left.\begin{matrix} y_{\text{left}}^{\text{LO}} \\ y_{\text{straight}}^{\text{LO}} \\ y_{\text{right}}^{\text{LO}} \end{matrix}\right\} \overset{sum}{\Longrightarrow} \sum y_i^{\text{LO}} = 1$

To translate these predictions to a steering command, we use a controller that computes a turning angle $\dot{a}_d$ counterclockwise:

$$\dot{a}_d = k_{\text{HD}} \cdot \left(y_{\text{right}}^{\text{HD}} - y_{\text{left}}^{\text{HD}}\right) + k_{\text{LO}} \cdot \left(y_{\text{right}}^{\text{LO}} - y_{\text{left}}^{\text{LO}}\right)$$

It is dependent to the weighted sum of differences between probabilities of the left and right turn commands for head direction control and lateral offset control respectively. k_{HD} and k_{LO} are positive parameters for adjusting the turning speed and are set to 10^0. It gives a negative angle for turning left and a positive for turning right.

As a final step, we compute the destination waypoint P_d relative to MAV's current position.

$$P_d = \begin{bmatrix} P_{d,x} \\ P_{d,y} \end{bmatrix} = \begin{bmatrix} P_{c,x} + v_c \cdot \cos a_d \, dt \\ P_{c,y} + v_c \cdot \sin a_d \, dt \end{bmatrix} \text{ with } a_d = a_c + \dot{a}_d$$

Table 3: Waypoint Function Parameters

Waypoint Function Parameter	Description
$P_{d,x}, P_{d,y}$	x, y coordinates of desired position waypoint
$P_{c,x}, P_{c,y}$	x, y coordinates of current position waypoint
v_c	Fixed desired forward speed
dt	Time sampling
a_c	Current heading angle
a_d	Desired heading angle

In Table 3 the waypoint function parameters are given. When the turning angle is given, the destination waypoint P_d can be computed for a given time sampling e.g. 1 sec, and a fixed forward speed e.g. 1 m/s. Then, the vehicle is oriented towards the new waypoint direction. The computed waypoint is sent to MAV's flight controller for flight plan execution.

2.3 Environmental Awareness

One important task in autonomous navigation is to ensure safety during flight. A low flying MAV under the forest canopy needs to be environmentally aware of possible obstacles in close range, in order to avoid them. Professional solutions may offer complete stand-alone obstacle avoidance systems, based on visual and ultrasound sensors. This can be enhanced by the parallel use of object detection DNN which may detect objects not seen from the obstacle avoidance system in first place. Moving objects, like people or animals, is such a case.

2.3.1 Object Detection

In object detection algorithms, the main goal is to detect and localize with a bounding box (BB) the item of interest inside an image. This can be done for multiple items representing different objects, and their BBs may overlap. Each of the object detection DNN architecture performs differently. We need to focus on objects that are usually found inside or near a forest trail path. Through a literature review, we decided to examine the most relevant and competitive DNN architectures for object detection, running on the Jetson TX2 on-board embedded GPU system.

2.3.1.1 Legacy Networks

- **R-CNN**: The Region based Convolutional Neural Network was introduced by Girshick et al. (2014) [25] and was the first model that was able to identify multiple occurrences of the same object within an image. Unlikely, it is computationally source demanding and cannot perform on embedded systems in real-time.

- **Fast R-CNN**: An improved version of the same algorithm was published by Girshick (2015) [26] in the following year. It constitutes a faster variant of the previous version, but it lacks on performance.
- **Faster R-CNN**: Both previous networks used the selective search method (Gandhi [29]), to determine the region proposals, which made them inefficient and computationally expensive. Region Proposal Networks (RPN) introduced by Shaoqing et al. (2016) [27] replaces selective search method by generating the region proposals and the BBs directly within the image. Faster R-CNN (2016) is based on Fast R-CNN combined with RPN, which is trained to generate high quality region proposals, resulting in improved performance and accelerated computation process.
- **SSD**: Single-Shot Detector (2016) [33] is a one stage detector, in contrast to the aforementioned Faster R-CNN and its variants. It simultaneously predicts the BB and the class as it processes the image. It is more accurate than Faster R-CNN and provides enormous speed gains.
- **YOLO v2 and v3**: YOLO v2 (You Only Look Once) was introduced by J. Redmon et al. (2016) [31] as a one stage detector. In contrast to other CNN algorithms which use region proposals for object localization; this one predicts BBs and class probabilities using a single convolutional network. It divides images into grid cells and predicts BBs using dimension clusters as anchor boxes. In 2018, an updated version 3 [32] was published by the same author. As a backbone network, yolov3 uses Darknet-53 for feature extraction, which is a residual network consisted of 53 convolutional layers. BBs are predicted in 3 different scales through extracting features from these scales. This version has an improved speed and detection accuracy over small-sized objects in contrast to previous one. There is also a tiny port that uses the same concepts but with a degraded total of convolutional layers and only 2 scales, leading to much greater inference speed with a cost on accuracy.

2.3.1.2 State-Of-The-Art Networks

- **MobileNetv2-SSDLite**: MobileNetv1 is an efficient CNN model introduced by Google in 2017. It uses a modified version of regular SSD. In SSDLite, regular convolutions are replaced by depth-wise separable convolution layers, which reduce the model size and the complexity cost of the network. This gives a great compatibility for embedded systems, such as mobile devices, because it makes it lightweight with high FPS rate. In 2018 MobileNetv2 [40] was introduced. It is an improved module with inverted residual structure. It is 20x more efficient and 10x smaller than YOLOv2.
- **YOLO v4**: YOLOv4 was introduced by Bochkovskiy et al. [35] in 2020. It has an improved performance over the previous version. It uses a new backbone named CSPDarknet-53, which is a Cross Stage Partial (CSP) network, adding the use of Spatial Pyramid Pooling (SPP), Path Aggregation Network (PAN), and mosaic data augmentation method. Unfortunately it lacks accuracy when dealing with numerous small objects in the scene. A tiny port is also available for low power embedded systems.

2.3.1.3 Performance Comparison

Several researchers have been improving their neural networks, seeking higher precisions in object detection accompanied by near real time performance. For precision evaluation, a standard dataset is used and the common metric is the mean Average Precision (mAP). It provides the mean value of the average precision generated from all the class objects within the dataset challenge. For performance evaluation, the metric used is frames per second (FPS). We assume that a near real time speed can be achieved between 15 and 30 fps and a real time speed with above 30 fps. mAP is inversely proportional to FPS speed, so better accuracy usually comes with slower FPS.

In Table 4 and Table 5, performance results are shown of selected algorithms used for object detection. All metrics have been sourced from their initial publications. For each architecture we provide: the input image dimensions, the backbone used, the mAP and AP-50 metrics of Ms COCO [38] 2017 dataset (80 classes), and the FPS on Jetson TX2. From [39], FPS on YOLO v3 and v4 variants is given for Jetson Nano. We assumed that TX2 is on average 2.5x faster than Nano.

TensorRT is a framework provided by NVIDIA and written in CUDA for optimizing deep learning models running on embedded systems such as Jetson TX2. By using TesnorRT, precision data type of model's weights and parameters can be reduced, thus inference can be performed at half precision floating point (FP16). This is a smart technique for increasing model's speed with exchange a small degradation of the accuracy. For some of the selected models, an optimized version is given, that is converted into TensorRT format with FP16 precision. For YOLO models we observe that TensorRT engine runs at **~4.2 times** the speed of the original Darknet model.

Table 4: Performance comparison of object detection DNNs (Legacy Networks)

Network Architecture	Backbone	mAP @[.5,.95]	AP-50 @.5	FPS on TX2
Faster R-CNN [27]	ResNet-101 [24]	27.2	48.4	0.9
SSD-512 [33]	ResNet-101-SSD	31.2	50.4	11~12
MobileNet-v1-SSDLite [40]		22.2	–	~21
YOLOv2-608 [31]	Darknet-19 [31]	21.6	44.0	7
YOLOv2 tiny [31] *Pascal VOC	Darknet-19	–	–	15~16
YOLOv3-288 [37]	with TensorRT 7 (FP16 precision)	33.1	60.1	~20.4
YOLOv3-416 [37]	with TensorRT 7 (FP16 precision)	37.3	66.4	~12.3
YOLOv3-608 [37]	with TensorRT 7 (FP16 precision)	37.6	66.5	~6.3
YOLOv3-tiny-416 [32]	Darknet-53	16.6	33.1	12
	with TensorRT 7 (FP16 precision)			37

Table 5: Performance comparison of object detection DNNs (State-Of-The-Art Networks)

Network Architecture	**Backbone**	**mAP @[.5,.95]**	**FPS on TX2**
MobileNet-v2-SSDLite **[40]**		**22.1**	**~28**
YOLOv4-288 [37]	CSPDarknet53	37.1	~4.7
	with TensorRT 7 (FP16 precision)		~19.8
YOLOv4-416 **[37]**	CSPDarknet53	**45.3**	~2.75
	with TensorRT 7 (FP16 precision)		**~11.5**
YOLOv4-608 [37]	CSPDarknet53	48.3	~1.4
	with TensorRT 7 (FP16 precision)		~5.88
YOLOv4-tiny-416 **[39]**	With TesorRT 7 (FP16 precision)	**19.6**	**64**
Scaled-YOLOv4-512 [39]	With TesorRT 7 (FP16 precision)	43.6	10

After evaluation, we ended up using YOLOv4-416 for high accuracy, and YOLOv4-tiny-416 or MobileNetv2-SSDLite for speed. For an optimized version of these networks running on NVIDIA Jetson board, we choose tkDNN [41], that is a Deep Neural Network library built with cuDNN and tensorRT primitives. The main goal of tkDNN project is to exploit NVIDIA boards as much as possible to obtain the best inference performance.

2.3.2 Obstacle Avoidance System – «Guidance»

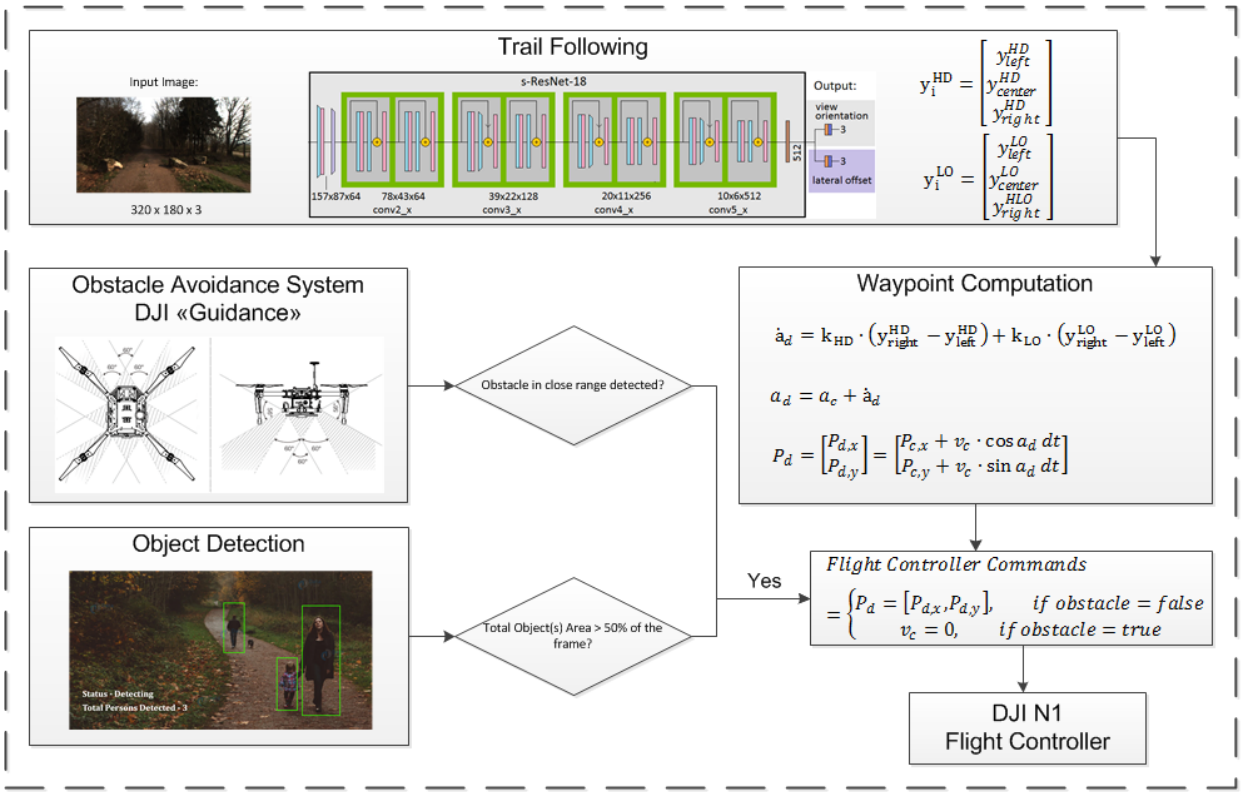
The DJI Matrice 100 framework that we use comes with an integrated collision and obstacle avoidance system called «Guidance». DJI's Guidance system [42] is a sensor-based navigation aid that can be installed on flying platforms or any other carriers. It uses ultrasonic sensors and stereo cameras to gather real time data about its surroundings. These sensors are equally shared at five sticks: four to cover all horizontal directions for obstacle avoidance plus a fifth aimed at the ground for X/Y positioning. Guidance Core collects sensor data, such as velocity, obstacle distance, position, IMU readings (acceleration, attitude, etc.), and in continue feeds the processed data to the DJI flight control system or to other intelligent systems of the carrier. Aircrafts equipped with Guidance system are able to perform hovering and obstacle sensing functions in GPS-denied environments.

Image 1: DJI's Guidance Obstacle Avoidance System [42]

2.4 Algorithms Integration

The framework of algorithm integration is designed as shown in Figure 4. At first, the current frame is fed in parallel to all three algorithms for processing. Trail navigation algorithm gives as an output, 6 percentages of head direction and lateral offset parameters. Through waypoint computation controller, these are translated into coordinates of the desired destination position. If there is no obstacle in its path, the desired waypoint is forwarded to MAV's flight controller for flight plan execution.

The other two algorithms are used for obstacle avoidance. DJI's «Guidance» system explores the surrounding area and if an obstacle is found within a close range, it overrides the waypoint commands and immediately stops and hover. In order to strengthen its weakness in detecting moving obstacles on time, object detection algorithm was added. It is used to detect an object by providing its localization through a BB, followed by the label of the category inferred and the confidence percentage. If the box exceeds a threshold percentage of the image, i.e. 50%, then the waypoint commands are overridden and cause the MAV to stop and hover.



$$y_i^{HD} = \begin{bmatrix} y_{left}^{HD} \\ y_{center}^{HD} \\ y_{right}^{HD} \end{bmatrix} \qquad y_i^{LO} = \begin{bmatrix} y_{left}^{LO} \\ y_{center}^{LO} \\ y_{right}^{HLO} \end{bmatrix}$$

$$\dot{a}_d = k_{HD} \cdot (y_{right}^{HD} - y_{left}^{HD}) + k_{LO} \cdot (y_{right}^{LO} - y_{left}^{LO})$$

$$a_d = a_c + \dot{a}_d$$

$$P_d = \begin{bmatrix} P_{d,x} \\ P_{d,y} \end{bmatrix} = \begin{bmatrix} P_{c,x} + v_c \cdot \cos a_d \, dt \\ P_{c,y} + v_c \cdot \sin a_d \, dt \end{bmatrix}$$

$$= \begin{cases} P_d = [P_{d,x}, P_{d,y}], & if\ obstacle = false \\ v_c = 0, & if\ obstacle = true \end{cases}$$

Figure 4: Algorithm integration framework

There is also a controller for teleoperation commands that can override at any time the trail following DNN algorithm. This can be used for emergency situation or in training session.

3. HARDWARE ARCHITECTURE

In order to set up a robust and flexible platform, flying under the forest canopy, we had to make a synthesis from different parts and accessories. Below, we present a detailed description of all the hardware used, including the setup procedure that was followed.

3.1 Hardware System Overview

The MAV that we used in our experiments is shown in Image 2. It uses as baseline platform the DJI's Matrice 100 (M100) quadcopter. It is accompanied with some hardware modules critical for its flight capability: N1 flight controller, Zenmuse Z3 camera with gimbal supporting resolutions up to 4K video @ 30 fps, C1 remote controller with an operating range of up to 5 km, and a visual guidance system for obstacle avoidance. In addition, it is also equipped with NVIDIA's Jetson TX2 supercomputer, which is used for vision processing and other computational demanding tasks (ROS, Wi-Fi Network Connectivity, etc.).

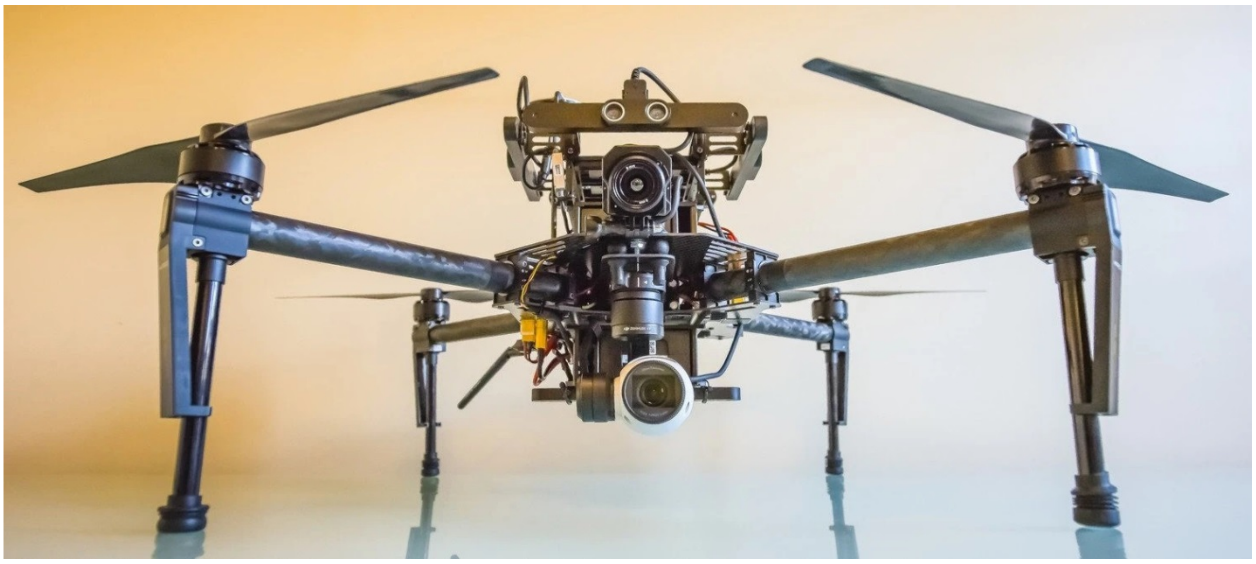


Image 2: Overview of our custom MAV framework

A hardware overview of our MAV system is shown in Figure 5. It is consisted of the following sub-sections:

- Power Distribution Circuity (PDC) that is seated onto the center frame of the platform. It is responsible for the correct power management of its components.
- Flight Controller (FC) that is responsible for the rotors movement and the command navigation execution. It has an embedded processor that receives and processes data from sensors such as GPS, IMU, and gimbal with Camera. It also offers external communication through RF antennas.
- Actuators that are 2 front and 2 rear rotors controlling propellers speed.
- External Peripherals that could be the official C1 Remote Control or a third party joypad used for manual navigation.

- «Guidance» System that is an optional module mounted on the baseline platform supporting data transfer to/from Flight Controller. It is used for obstacle avoidance.
- External Computer Module that is used for autonomous navigation. In our case we use NVIDIA’s Jetson TX2 supercomputer that handles process demanding vision-based DNN algorithms. It also offers a wireless connection through Wi-Fi module and carries a wide angle camera with 4K resolution. It exchanges data with N1 FC.

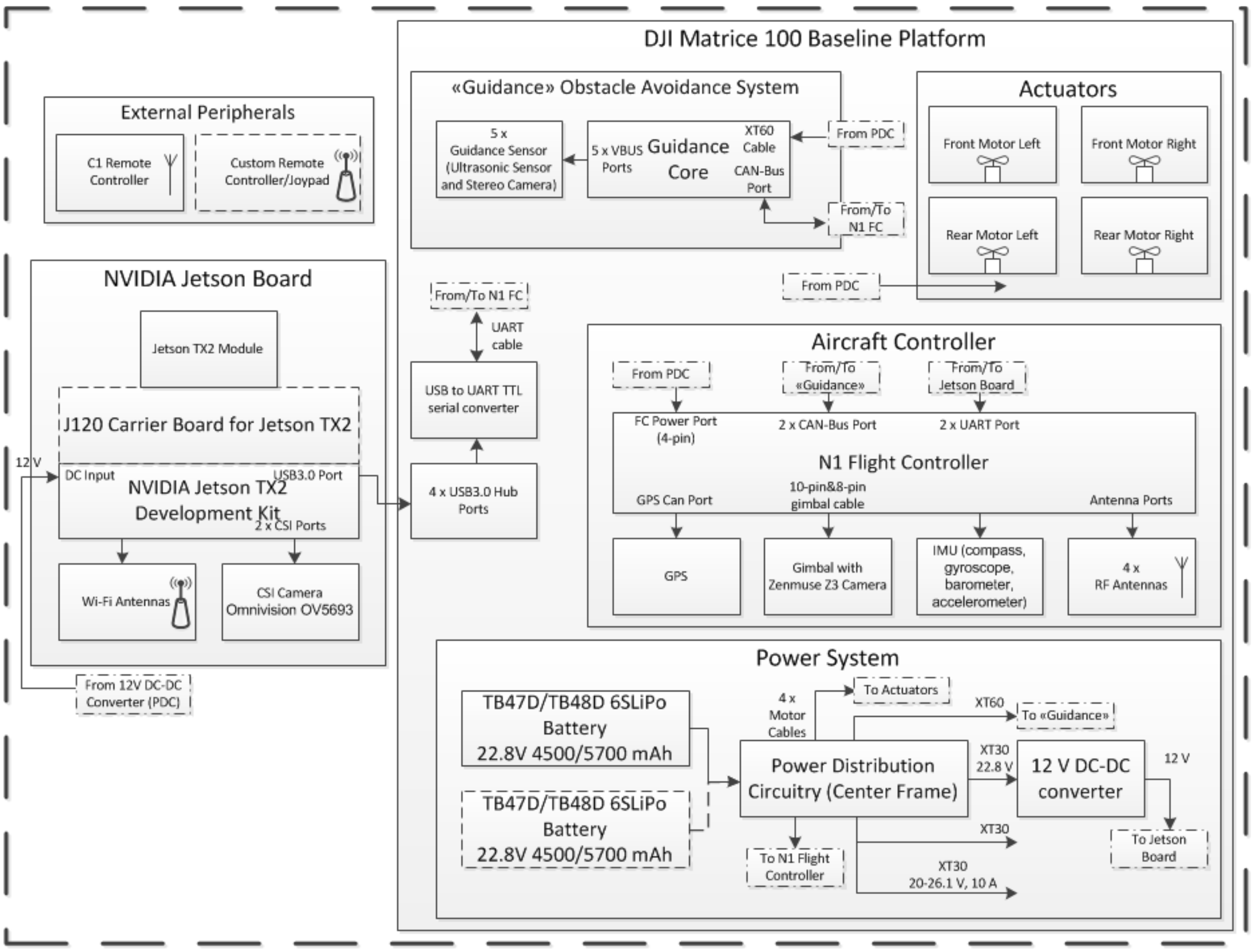


Figure 5: System Hardware Block Diagram

In continue we present a more detailed breakdown of hardware parts.

3.1.1 DJI Matrice 100 (M100) quadcopter

The Matrice 100 [Image 3] is a stable, flexible, and powerful flying platform, designed and produced by Da Jiang Innovation (DJI) Science and Technology Company Ltd, and released for first time in 2015. It was DJI’s first fully-integrated UAV platform designed for light duty commercial applications, competing small- and mid-sized drones of its class. It offers great customizability by allowing developers to make modifications according to their specific needs. It has 2 expansion bays that can be configured to carry any set of sensors or devices (up to 1 kg). Through DJI SDK, developers have full control over the platform with the built-in API Control feature, allowing them to build custom mobile apps and advanced flight controls for any requirement. It can carry up to 2 Intelligent Flight Batteries that would extend the flight time to 40 minutes. The default package of M100 flying platform includes a flight controller, propulsion system, flexible cargo bays, GPS, dedicated remote controller, a great mobile app and a rechargeable battery.

Image 3: DJI Matrice 100 (M100) baseline platform [43]

The M100's N1 flight controller [Image 5] receives and processes the data from the local sensor suite (i.e., gyroscope, compass, barometer) and GPS receiver prior to sending the control information to each individual motor via its electronic speed control (ESC) circuits, which are designed to control the motor's thrust, revolutions per minute (RPM), and direction.

Image 4: DJI C1 Remote Controller [43]

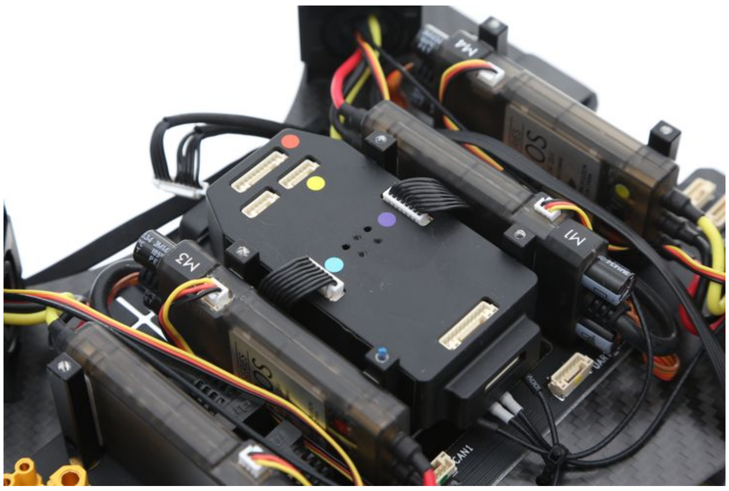


Image 5: DJI N1 Flight Controller [43]

In order for our robot to get a reliable position estimate, we are using DJI's default global positioning system (GPS) [Image 6] to read the robot's latitude and longitude coordinates.

Image 6: DJI's M100 GPS Module [43]

Image 7: DJI Guidance Sensors and Computing Core [42]

The M100 also contains a gimbal camera (Zenmuse Z3) [Image 8Image 6Image 8], which provides video feed to the human operator via the radio frequency (RF) up/down link. The movement of the M100 quadcopter UAV originates from the remote control stick [Image 4]; the signals are sent via RF data up/down link and are passed to the N1 flight controller to execute the desired movement by directing the ESC and motors to increase or decrease speed.

The movements supported for the M100 are divided into two categories: vertical and horizontal axes. In the vertical plane, the UAV is designed to: Hover (stays steady in the sky), Ascend (increases its altitude), and Descend (reduces its altitude). In the horizontal plane, the UAV is designed to: Yaw (rotate to the left or the right), Pitch (moves forward or backward), and Roll (moves sideways left or right).

Image 8: DJI Zenmuse Z3 Camera [47]

Image 9: Intelligent Flight Battery TB47D/TB48D [43]

The M100 is integrated with a Zenmuse Z3 gimbal camera system for live video feed during the flight. This Z3 camera contains a Sony complementary metal oxide semi-conductor (CMOS) sensor with 12.4M pixels, which provides 4K / FHD / HD quality video recording to the user. The three-axis gimbal controller receives data from the N1 flight controller to compute the required angular motion correction to the camera for video stabilization during flight, and to make control changes to point the camera according to user-defined inputs.

Table 7 summarizes the key specifications of the DJI Zenmuse Z3 gimbal camera.

Table 6: Key Specifications of the Matrice 100 MAV [43]

Parameters	Values
Performance	
Hovering Accuracy (P-Mode with GPS)	Vertical: 0.5m, Horizontal: 2.5m
Max. Angular Velocity	Pitch: 300o/s , Yaw: 150o/s
Max. Tilt Angle	35o
Max. Speed of Ascent	5 m/s
Max. Speed of Descent	4 m/s
Max. Wind Resistance	10 m/s
Max. Speed	22 m/s (ATTI mode, no payload) 17 m/s (GPS mode, no payload)
Battery Voltage/Capacity	TB47D : 22.8V / 4500 mAh TB48D : 22.8V / 5700 mAh
Hovering Time w/o payload **(with Zenmuse Z3)**	19 mins with TB47D 23 mins with TB48D
RF Data Up/Down Link	
Operating Frequency	5.725 ~ 5.825 GHz (Video) 2.400 ~ 2.483 GHz (Data)
Estimated Transmission Distance **(Line-of-sight)**	CE: 3.5 km FCC: 5 km
Structure	
Diagonal Wheelbase	650 mm
System Weight	2355 g with TB47D 2431 g with TB48D
Maximum Takeoff Weight	3600 g
Expansion Bay Weight	45 g
Zenmuse Z3 Gimbal Camera	247 g

Table 7: DJI Zenmuse Z3 Gimbal Camera Specifications

Parameters	Values
Model	Zenmuse Z3 (FC250)
Sensor	Sony EXMOR 1 / 2.3" CMOS
Shutter Type	Global Shutter
Lens	Field of View (FOV): 94o Focal Length (35 mm Equivalent): 20 mm Aperture: F/2.8
Video Recording	UHD (4K): 4096 x 2160 3840 x 2160: FHD (1080p): 1920 x 1080 HD (720p) 1280 x 720
File Format	Photo: JPEG, DNG Video:MP4 in .MOV
Photography Modes	Storage on MicroSD Card Single Shot, Burst (3, 5, 7 frames per sec)
Interface	Proprietary of DJI. Undisclosed.

3.1.2 Power Supply and Consumption

The M100 system is powered by a single TB47D / TB48D [Image 9], 6S LiPo battery with voltage rated at 22.8V, and capacity of 4500 mAh / 5700 mAh. The M100 has its own power distribution circuitry to provide regulated power to the base unit hardware modules. The M100 power circuitry has two additional ports for 22.8V unregulated voltage, which can be supplied to additional experimental hardware units.

The Jetson Development board contains a 5 V DC-DC converter capable of powering most sensors, so the only components which require direct power from the main power supply is the TX2 itself. The selected Lithium Polymer battery has 6 cells in series, meaning the nominal battery voltage is 22.8 V when fully charged. This means that a 12 V output buck converter is required to step down the DC voltage to a level safe for the TX2. This approach was followed by Cookson et al. (2019) [51] for the development of a UGV taking part on 2019 Intelligent Ground Vehicle Competition.

An indicative estimation of peripheral power consumption and battery operation time is shown in Table 8 and

Table 9. We use the Ohm's law that says power **P** of an electrical device is equal to voltage **V** multiplied by current **I**: $P = V * I$. Also amp hours are a measure of electric charge **Q** (the battery capacity): $E = V * Q$. As energy **E** is power **P** multiplied by time **T**: $E = P * T$. All we have to do to find the energy stored in a battery is to multiply both sides of the equation by time: $E = V * I * T$.

In Table 8, we give details on our computations. We need to estimate the power consumption (Watts/hour) of each of the 3 main components: (i) TX2 with all its sensors, (ii) DJI Guidance, and (iii) M100 with 1 kg payload and Zenmuse Z3 camera. For the first two, we have 99 Wh and 12 Wh given from their technical specifications. For M100 we have to make the maths. By using one TB48D battery, we know that its max power is 130 Wh. We also have the max hovering time for cases (1), (2), and (3), so we can calculate the Wh respectively. Utilizing the Wh for the first three cases, we can estimate the Wh for the (4) case, which is the power consumption per hour for the M100. At last we calculate the total operation time by dividing the battery energy (130 Wh) with the sum of the 3 main components power consumption (659.1 Wh).

Table 8: Estimated Peripheral Power Consumption

Components	Voltage (V)	Current (mA)	Power (W)
TX2 Dev Kit (B04) with: **- TX2 Module** **- OmniVision OV5693 (5MP)**	19	4740	90 + (10% margin) * 90 = **99** (Given from TX2 specs)
M100 with: **- GPS** **- Zenmuse Z3** **- N1 FC** **- 1 kg payload**	N/A	5700	493.29+(10% margin) * 493.29 = **548.1** (Computed in (4))
(1) Hovering Time[2] (with TB48D) no payload: 28 min			$60 * \frac{130}{28} = 278.5$
(2) Hovering Time (with TB48D) with 1 kg payload: 16 min			$60 * \frac{130}{16} = 487.5$
(3) Hovering Time (with TB48D and Zenmuse Z3) no payload: 23 min			$60 * \frac{130}{23} = 339.1$
(4) Hovering Time (with TB48D and Zenmuse Z3) with 1 kg payload: **N/A min, Power Consumption estimation from (3)-(1)+(2)**			339.1-278.5=60.6 for Zenmuse Z3 + 487.5 = 548.1
DJI Guidance	11.1 ~ 25	N/A	**12** (Given from specs)
DJI C1 RC	–	–	–
Total Power with 10% margin (W)			659.1

[2] The hovering time is based on flying at 10m above sea level in a no-wind environment and landing with 10% battery level

Table 9: Estimated Battery Operation Time

Total Power Consumption (W) with 10% margin	659.1
TB48D Battery Energy (Wh)	130
TB48D Battery Voltage (V)	22.8
TB48D Battery Capacity (mAh)	5700
Operation Time (minutes)	$\frac{130}{659.1} * 60 = \mathbf{11.8}$

The total time of operation for our custom framework would last for approximately 11.8 minutes if we use the TB48D Battery, or twice the time i.e. 23.6 min if we use two TD48D batteries. We assumed that the framework flies with a max payload of 1 kg and Zenmuse Z3 camera onboard. In our computations we included a landing 10% margin.

3.1.3 NVIDIA Jetson TX2 Development Kit

The main computer used for vision-based processing is the NVIDIA Jetson TX2 [44]. The Jetson is an embedded system-on-module (SoM) with dual NVIDIA Denver2 and quad-core ARM Cortex-A57, 8GB 128-bit RAM and integrated 256-core Pascal GPU. This platform is ideal for mobile robotics research due to its small size and powerful processing capabilities. We use the Jetson, along with the Jetson Development board, as the central computer for our MAV. We are using the Development board so the Jetson can connect to a Wi-Fi network without additional peripherals, and to allow easier access to the built in general purpose input/output (GPIO) pins. The Jetson will be responsible for reading in all of the sensor data, running all ROS nodes, computer vision algorithms and sending telemetry data through serial connection to the N1 flight controller.

Image 10: NVIDIA Jetson TX2 Development Kit [44]

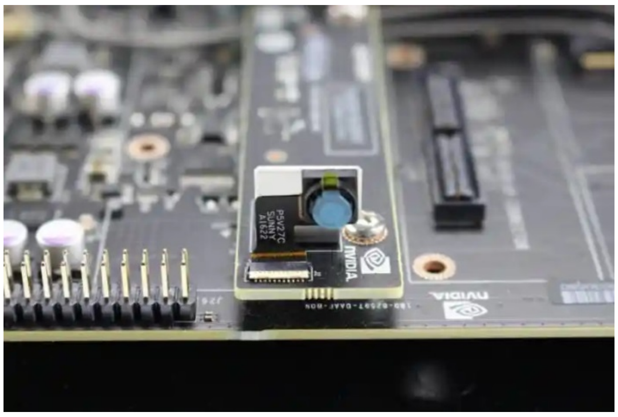


Image 11: Jetson TX2 Camera Module [45]

As video input for our vision-based algorithms, we use the Jetson's CSI camera module Omnivision OV5693 [Image 11], loaded with a 5MP image sensor, delivering DSC quality imaging and low-light performance as well as full 1080p high-definition video recording at 30 fps.

The connection between TX2 and N1 flight controller is done through a USB to TTL serial converter [Image 13]. We connect the UART TTL output from the flight controller to the converter and then to a usb port of TX2 board.

Table 10: Key Specifications of Jetson TX2 module

Characteristic	TX2
GPU	NVidia Pascal™ architecture with 256 NVidia CUDA cores
CPU	Dual-core Denver 2 64-bit CPU and quad-core ARM A57 complex
Memory	8 GB 128-bit LPDDR4
Storage	32 GB eMMC 5.1
Video Encode	Up to 2 x 4K @ 30 fps
Video Decode	Up to 2 x 4K @ 30 fps, 12-bit support
JetPack support	Jetpack 4.5

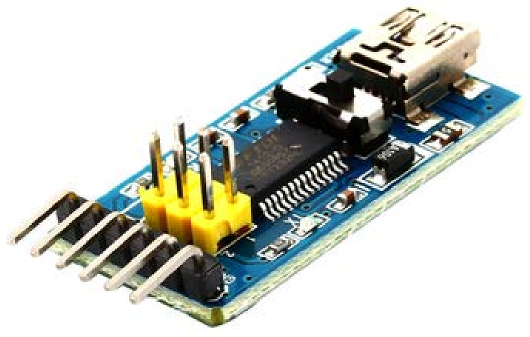


Image 13: USB to TTL Serial Converter

Image 12: Auvidea J120A-IMU/MCU carrier for Jetson TX1/TX2 [46]

A better approach is to replace TX2 dev kit with J120 carrier board [Image 12] for a more lightweight solution.

4. SOFTWARE ARCHITECTURE

In this chapter we present the software used for our MAV build. Emphasis is given in the subsystems which are represented by ROS nodes.

4.1 Software System Overview

Software configuration can be broken down into modules residing within the N1 flight controller and the NVIDIA Jetson TX2 developer kit.

N1 flight controller contains the following modules: (a) flight computer, (b) inertial measurement unit (IMU) including compass, gyroscope, barometer, accelerometer, and (c) GPS.

Jetson TX2 is used as an external onboard computer mounted on the aircraft, and it connects to the flight controller through a direct serial (UART) connection. It provides automation of flight through the following modules: (a) trail following model, (b) object detection model, (c) camera feed, and (d) waypoint computation.

4.2 Jetpack (Linux4Tegra-L4T)

Jetpack[3] is a suite of useful tools for building AI applications on top of Jetson TX2. It includes a Linux OS image for Jetson products, along with libraries and APIs, samples, developer tools, and documentation. Some tools are used directly on a Jetson system, and others run on a Linux host computer connected to a Jetson system.

JetPack libraries and APIs include:

- TensorRT and cuDNN for high-performance deep learning applications
- CUDA for GPU accelerated applications across multiple domains
- NVIDIA Container Runtime for containerized GPU accelerated applications
- Sample applications and other libraries for visual computing tasks.

For our experiments we used Jetpack version 4.5 that comes with a customized Linux distro named Linux4Tegra (L4T) 32.5.0.

4.3 DJI's Onboard SDK (OSDK)

The communication between TX2 and N1 FC is done through the DJI's open-source software library, the Onboard SDK[4] [49]. The SDK gives access to aircraft telemetry, flight control and other aircraft functions, meaning a developer can use the SDK to control flight. The SDK comes with a fully featured ROS wrapper compatible with ROS standards. The latest compatible version for M100 is 3.9.0.

[3] https://developer.nvidia.com/embedded/jetpack

[4] https://github.com/dji-sdk/Onboard-SDK

Figure 6 illustrates how the DJI Onboard SDK is used into an application, and how it is connected to a DJI aircraft.

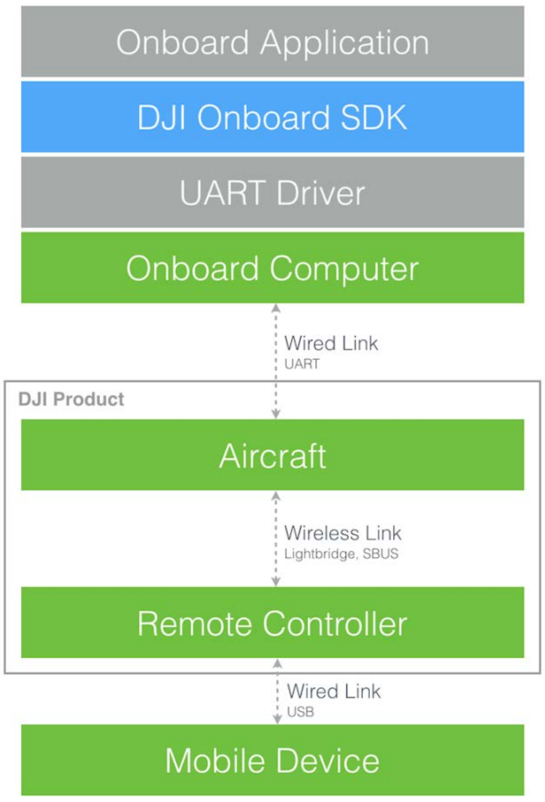


Figure 6: Connection to User Application and DJI's Aircraft [49]

4.4 ROS

For our algorithms implementation, we chose to use ROS [50], an open source operating system for robots. ROS can be characterized as a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviours [52].

In our project we use ROS Noetic. It provides the environment in which the DJI Onboard SDK, and each of the vision-based algorithms, will be run as a service. All these services are run as ROS nodes.

4.4.1 ROS Nodes

A ROS node is a process that interacts with ROS network and completes the tasks given. It is identified by a unique name and communicates with other nodes through topics. In our setup, each node corresponds to a single function and only. So in total we have ROS wrappers for (a) camera feed, (b) trail following DNN, (c) object detection DNN, (d) waypoint computation, (e) manual Remote Control, and (f) video frame splitter.

Below we give the ROS nodes dedicated for each of our services.

4.4.1.1 Camera node

The camera node `gscam` publishes to the `/camera/image_raw` topic using the standard ROS Image message. This is the master source which provides information to other nodes like the trail following DNN and object detection DNN.

4.4.1.2 DNN node

The `caffe_ros` node provides support for DNN inference using TensorRT library. This node currently supports Caffe models such as classification, YOLO-based object detection, regression and semantic segmentation. It produces messages in different formats depending on the type of the network and post-processing options (e.g. TrailFollowing vs YOLO)

1. TrailFollowing DNN

   This node publishes output of the DNN (e.g. softmax layer) using standard Image message. The default topic name is `/caffe_ros/network/output`.
   The node requires two parameters:

   a. `prototxt_path` - path to the Caffe model .prototxt file

   b. `model_path` - path to the Caffe model binary file. In addition, other parameters like input/output layer names (`input_layer`/`output_layer`) have to be set correctly. A command line example would be:

```
rosrun caffe_ros caffe_ros_node __name:=trails_dnn
_prototxt_path:=/data/src/autonomous_drones/models/pretrained/Aria
dneNet_SResNet-18.prototxt
_model_path:=/data/src/autonomous_drones/models/pretrained/Ariadne
Net_SResNet-18.caffemodel _output_layer:=out
```

2. Object detection DNN (YOLO, Mobilnet)

   In order to apply an object detection functionality as a post-processing procedure, we have to launch the node with `post_proc:=YOLO` argument. The node publishes output of the DNN using standard Image message in a certain format: the output is a 2D, single-channel "image" that has the following format: `WxHx1` (so encoding == `32FC1`) where `W` is fixed and equals 6, and `H` is equal to the number of detected objects. For example, if the DNN has detected 2 objects, then the output is `6x2` image.
   For each detected object, the 6 values are the following:

```
0  : label (class) of the detected object (e.g. person or a dog).
1  : probability of this object.
2,3: x and y coordinates of the top left corner of the object in
image coordinates.
4,5: width and height of the object in image coordinates.
```

   All values are 32-bit floats, including label. Label indices correspond to 20 classes from PASCAL VOC 2012 dataset.

For the making of this node we were also inspired by the Advanced Sensing - Object Detection Sample[5].

4.4.1.3 Image Publisher node

The `image_pub` node is a simple ROS node that reads video or image files and publishes frames as a ROS topic as an Image message. There is one mandatory parameter, `img_path`, which specifies the path to an image or video file. This node supports setting custom frame rates as well as repeating an image indefinitely. The default topic is `/camera/image_raw` which can be changed using the `camera_topic` parameter.

An example of publishing frames from the video file using the file's native frame rate:

```
rosrun image_pub image_pub_node _img_path:=/data/videos/trail_test.mp4
```

An example of publishing the same image repeatedly at 30 frames / sec:

```
rosrun image_pub image_pub_node _img_path:=/data/images/trail_right.png
_pub_rate:=30 _repeat:=true
```

4.4.1.4 Onboard SDK Controller node

The `osdk_controller` node is a ROS node that takes as input the results of DNN nodes and then computes waypoints, in order to send them through OSDK connection to the N1 flight controller. The OSDK-ROS[6] was used. We were inspired from two different samples: Flight Control Sample[7] and GPS Mission Sample[8]. A `demo_flight_control`[9] sample was used for initial testing the connectivity between TX2 and N1 FC.

4.4.1.5 Guidance-SDK-ROS

The `guidance` node is a ROS node that processes the Guidance sensor inputs and searches for obstacles in a close perimeter. When an obstacle is inside this perimeter, it gives a Hover command to Flight Controller overriding any other commands. We used the official ROS package of Guidance SDK[10] for 32/64 bit Ubuntu.

[5]https://developer.dji.com/onboard-sdk/documentation/sample-doc/advanced-sensing-object-detection.html

[6] https://github.com/dji-sdk/Onboard-SDK-ROS

[7] https://developer.dji.com/onboard-sdk/documentation/sample-doc/flight-control.html

[8] https://developer.dji.com/onboard-sdk/documentation/sample-doc/missions.html

[9] https://developer.dji.com/onboard-sdk/documentation/development-workflow/sample-setup.html

[10] https://github.com/dji-sdk/Guidance-SDK-ROS

4.4.2 ROS Node Hierarchy

ROS nodes help us to break our software into smaller pieces, communicating each other in hierarchy. For a more comprehensive presentation, we give a hierarchy system software diagram, that categorizes services into layered subsystems. Subsystems are represented as ROS nodes. Each ROS node is categorized according to its functionality and connectivity to others. Lowe-level subsystems are usually nodes that read information from sensors and publish their data as ROS messages over topics, that in continue can be subscribed to higher level subsystems (ROS nodes). Higher level nodes receive the information from lower level nodes and use them as input to algorithms for final autonomous navigation decisions. Intermediate nodes combine multiple low level sources to produce derivatives needed from higher level nodes.

In Figure 7 the hierarchy of ROS nodes for our system is shown.

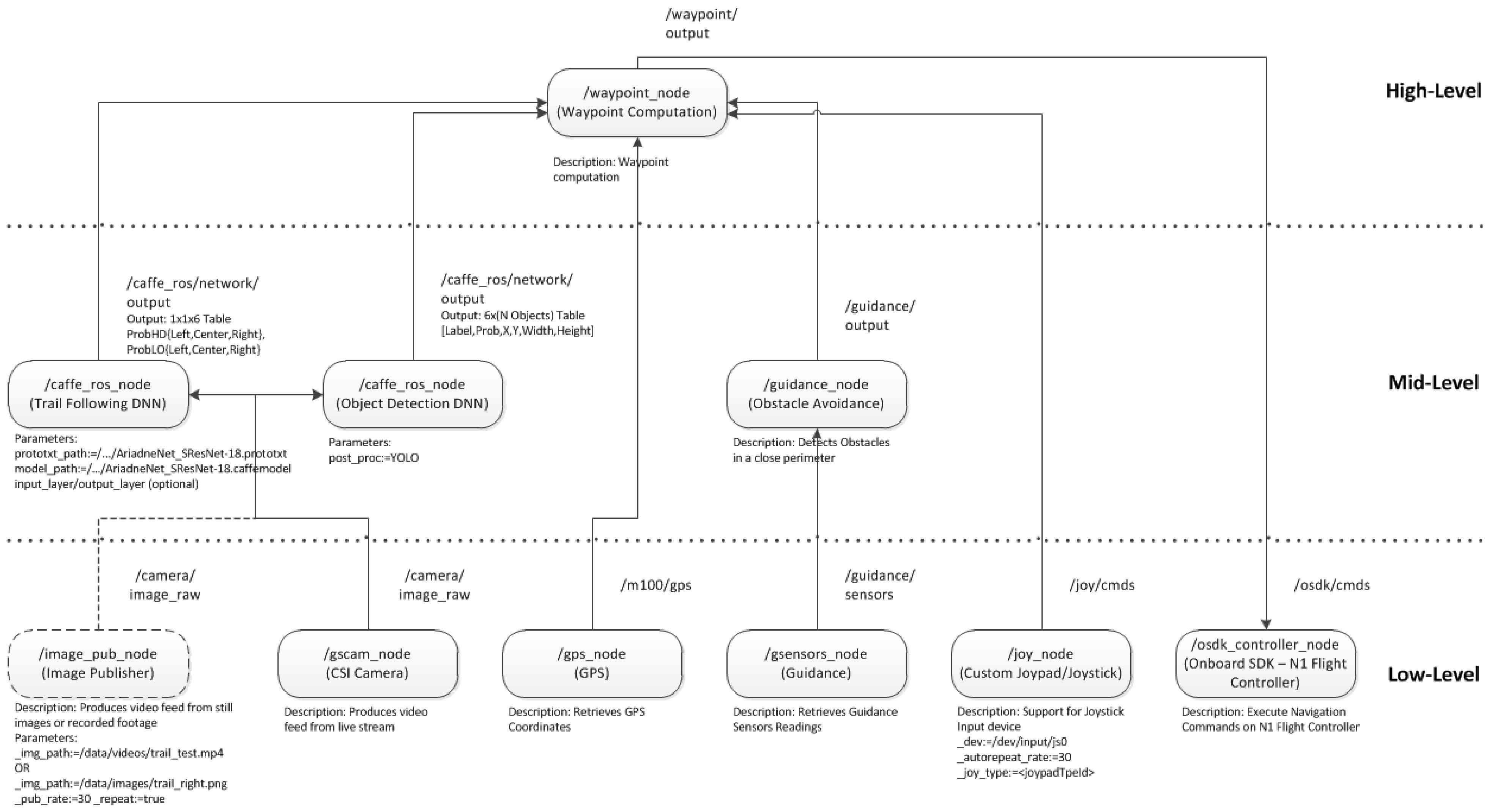

/waypoint/
output
/waypoint_node
(Waypoint Computation)
Description: Waypoint computation
High-Level
/caffe_ros/network/
output
Output: 1x1x6 Table
ProbHD{Left,Center,Right},
ProbLO{Left,Center,Right}
/caffe_ros/network/
output
Output: 6x(N Objects) Table
[Label,Prob,X,Y,Width,Height]
/guidance/
output
/caffe_ros_node
(Trail Following DNN)
/caffe_ros_node
(Object Detection DNN)
/guidance_node
(Obstacle Avoidance)
Mid-Level
Parameters:
prototxt_path:=/.../AriadneNet_SResNet-18.prototxt
model_path:=/.../AriadneNet_SResNet-18.caffemodel
input_layer/output_layer (optional)
Parameters:
post_proc:=YOLO
Description: Detects Obstacles in a close perimeter
/camera/
image_raw
/camera/
image_raw
/m100/gps
/guidance/
sensors
/joy/cmds
/osdk/cmds
/image_pub_node
(Image Publisher)
/gscam_node
(CSI Camera)
/gps_node
(GPS)
/gsensors_node
(Guidance)
/joy_node
(Custom Joypad/Joystick)
/osdk_controller_node
(Onboard SDK – N1 Flight Controller)
Low-Level
Description: Produces video feed from still images or recorded footage
Parameters:
_img_path:=/data/videos/trail_test.mp4
OR
_img_path:=/data/images/trail_right.png
_pub_rate:=30 _repeat:=true
Description: Produces video feed from live stream
Description: Retrieves GPS Coordinates
Description: Retrieves Guidance Sensors Readings
Description: Support for Joystick Input device
_dev:=/dev/input/js0
_autorepeat_rate:=30
_joy_type:=<joypadTpeId>
Description: Execute Navigation Commands on N1 Flight Controller

Figure 7: ROS Nodes Hierarchy Diagram

4.5 Docker Support

Docker is an open source software platform to create, deploy and manage virtualized application containers on a common operating system (OS), with an ecosystem of allied tools. All tools needed for implementing our system functionalities can be delivered through prebuilt Docker images containing ROS, OSDK, L4T, and ML libraries. The use of Docker constitutes an alternative solution to native installments on Linux4Tegra (L4T) OS (a modified version of Ubuntu 18.04). A detailed configuration guide can be found in Appendix III.

5. EXPERIMENTS

For a real flight experiment, we propose the framework described in the previous chapter. Due to compatibility issues we had in first place, as also access restrictions to the lab during the COVID-19 quarantine, we didn't manage to fly the drone stack in real-time environment. We propose two different types of experiments. For autonomous navigation, we suggest an experimental flight through a forest trail inside our campus. For object detection we did a comparison between object detection algorithms. We used an excerpt from a trail runner's recording found in FullHD on the internet. In the following sections, the results of these two experiments are presented.

5.1 Trail Following Experiment

One way to test our trail following model is to autonomously navigate through an almost straight 300 m forest trail path, with some minor turns, somewhere inside the campus. The trail has to be 2.5 to 3 m wide and the height of the MAV should be around 1.8 to 2 m. We must use a steady speed at 1 m/s, and the total time of test flight should have duration of 5.5 to 6 minutes. A challenge would be to override the autonomous flight 4 to 5 times by injecting rotations to the left or right, and if you see that the MAV always keeps returning to the center of the path, that would indicate a success to this experiment. As said previously, the trail following model and the object detection algorithm run on the Jetson TX2, utilizing its CPU and GPU at their max, while obstacle avoidance run in parallel as a separate service by Guidance core.

Table 11: Parameters for experiment

Parameter	Value
$\boldsymbol{k_{HD}}$	0.04
$\boldsymbol{k_{LO}}$	0.02
$\boldsymbol{v_c}$	1

5.2 Object Detection Experiment

As described in section 2.3.1.3, we explored the two most famous, state-of-the-art architectures, in object detection and their variants: SSD and YOLO. We used an excerpt from an online YouTube video [48] recorded from the sight of a trail runner. It is a trail run inside the Highbanks Metro Park, a metropolitan park in Central Ohio. It is named for its steep banks along the Olentangy River, the park's most unique feature[11].

We clipped the first 5 minutes of this virtual run and tested the two object detection algorithms. All architectures were run as an optimized version using TensorRT 7 (tkDNN framework). In the following table, we give an average of computed FPS on TX2. It seems to function at lower speeds than the ones referred in literature.

[11] https://en.wikipedia.org/wiki/Highbanks_Metro_Park

Table 12: Comparison of State-Of-The-Art DNN architectures for object detection

Network Architecture	Average FPS on TX2
YOLOv4-288	3
YOLOv4-608	1
YOLOv4-tiny-416	60
Scaled-YOLOv4-512	8
MobileNet-v2-SSDLite	25

6. CONCLUSIONS AND FUTURE WORK

6.1 Discussion on Results

This project presents an autonomous MAV system capable of trail following navigation using DNN based algorithms. It was built with parts of unknown compatibility status. The most challenging task was to overcome several incompatibilities between the Jetson TX2 with the N1 flight controller of DJI's Matrice 100 quadcopter.

The DNN architecture used for trail following was based upon existing work [1,2]. Initially, the IDSIA forest trail dataset [6] was used for training to enable the estimation of view orientation. Then, via transfer learning, our custom dataset was used to incorporate 3 additional categories for lateral offset estimation. This also improved the accuracy for inference in trails with local vegetation characteristics. With a quick look at performance results, trail DNN used gives the highest accuracy in comparison to other known trail following algorithms but it lacks in runtime speed, making it a secondary option for real time processing on an onboard computer such as Jetson TX2.

In the context of environmental awareness, two ROS nodes were used; one DNN algorithm for object detection (in order to detect known objects), and one for obstacle avoidance (in order to estimate depth). For object detection, several well-known DNN architectures were explored. YOLO was the most fast and accurate. For obstacle avoidance the «Guidance» system was used in default mode. Although it successfully detects obstacles of unknown type in real-time, it has significant delays for moving objects. This can be surpassed using object detection algorithms, such as YOLO.

6.2 Future Work

As future work, there are a few tasks that could be done in order to upgrade this project.

All the software applications used in this work can be integrated to an alternative hardware MAV platform in conjunction with a Pixhawk autopilot [20] (flight controller). A compatibility list of Pixhawk compatible airframe builds can be found at the online user guide documentation. Just for reference, a set of compatible multi-copters are: DJI Flame Wheel 450, Lumenier QAV250, DJI Matrice 100, Holybro S500.

Trail following algorithm that is based on supervised learning is limited to known and trained environments. It can be replaced by a deep reinforcement learning algorithm that stands out in unseen environments leading to more accurate estimations in trail navigation and obstacle avoidance.

In case of temporal distractions, such as wind disturbances, the MAV may miss its way causing the camera to miss the trail and resulting to inability on getting back to track. Therefore, autonomous navigation system needs a path recovery service to be run in the background, in order to guide the drone back to its trajectory.

A way of increasing accuracy in depth estimation would be to experiment with different sets of sensors. A set of numerous combinations of mono/stereo cameras with laser range finder (or additional other sensors) could be evaluated in terms of accuracy and efficiency. Relative neural networks and datasets for experiments should be used [19].

Support for manual teleoperation is provided through a remote controller. It is used for training purposes or emergency override. An innovative idea is to offer the same feature of remote teleoperation through an Augmented Reality application that would be running under a AR device (i.e. HoloLens). Implementation details can be found in projects [21, 22].

ABBREVIATIONS – ARCTICS – ACRONYMS

DNN	Deep Neural Network
IMU	Inertial Measurement Unit
GPS	Global Positioning System
IDSIA	Italian: Istituto Dalle Molle di Studi sull'Intelligenza Artificiale
ROS	Robot Operating System
MAV	Micro Aerial Vehicle
AR	Augmented Reality
FOV	Field Of View
mAP	mean Average Precision
FPS	Frames Per Second
BB	Bounding Box
DJI	Da Jiang Innovation
GPIO	General Purpose Input/Output
OSDK	DJI’s Onboard SDK
HD	Head Direction
LO	Lateral Offset
TL	Turn Left
GS	Go Straight
TR	Turn Right

APPENDIX I: Source Code Repository

The source code of this thesis, as also additional documentation, is available at https://github.com/gkalam/autonomous_drones

APPENDIX II: Dataset Preparation

II.A Dataset Creation for use on Lateral Offset Training

To train the lateral offset layers of the trail following DNN, we need to collect a three-sided simultaneous video, walking from the centre of the path. As a setup, we used a wide rig with 1 m baseline [Image 14]. Three full HD cameras are mounted on each edge and the centre, respectively. Each camera records a different front view, and the DNN learns to classify drone's lateral offset position relative to the centre line. It produces probabilities of being on the left side, right side or in the middle of the trail. All three cameras had same technical specifications: 120^0 horizontal FOV lenses and recording analysis in full HD (1920×1080) at 30 fps.

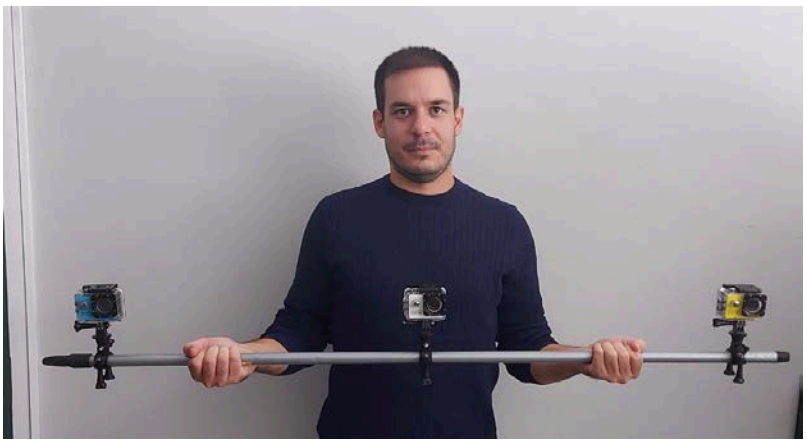


Image 14: Three camera wide baseline rig used for recording our dataset into Ilisia University Campus.

II.B Camera Calibration – Intrinsic Parameters

There is a difference at horizontal FOV between the camera used for inference on the MAV and the cameras used for gathering training data. The former usually uses lenses of 60^0 while the later uses lenses of 120^0. Wide angle lenses lead to fisheye distortion, which has to be removed prior to training procedure.

We use a camera calibration application in order to calculate intrinsic parameters of each camera. It is written in C++ and uses OpenCV library. Its first version was developed by [2] and used OpenCV3.0, while we ported it to support V4.0. We used the calibration target in landscape orientation as shown in Image 15, to capture several images (around 30) from different viewpoints with the wide angle cameras, in order to compute intrinsic camera parameters.

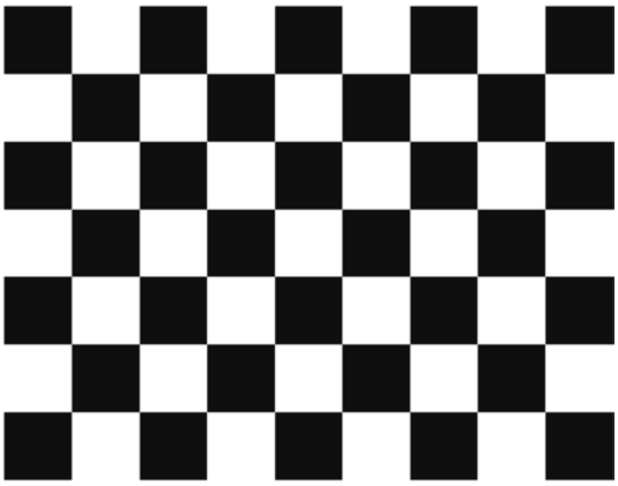


Image 15: **Chessboard Calibration Target – 8x6 Landscape Orientation**

Image 16: Result of undistorted image with checker board corners detected

In calibration, checkboard pattern is widely used because the corners of squares provide an easy to detect way of localizing them. They have sharp gradients in two directions, and also are at the intersection of checkboard lines. An indicative result of an undistorted image is shown in Image 16. The image should be sharp and should cover the whole target, in order for the frame to be ideal for selection.

After running the calibration application with our selected 30 image input, we get the intrinsic parameters as an yml file.

```
%YAML:1.0
---
Date: "Fri Jul 23 17:23:57 2021\n"
FrameWidth: 1920
FrameHeight: 1080
CameraMatrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1.7084553934190224e+03, 0., 8.7222499809355202e+02, 0.,
       1.2787870991072430e+03, 5.5679803246704967e+02, 0., 0., 1. ]
DistortionCoeffs: [ -2.3848078689012822e-01, 3.2889029218322099e+00,
    -2.0474728203992445e+01, 4.7210632074976793e+01 ]
```

Figure 8: Intrinsic parameters in .yml file format

II.C Data Pre-Processing: Frame Sampling, Undistorting, Virtual Views Extraction

For the need of DNN training, we split the video captures into single frames, and then select 1 frame (out of 30) for each second. For video splitting we use the python script «videoParserV3.py».

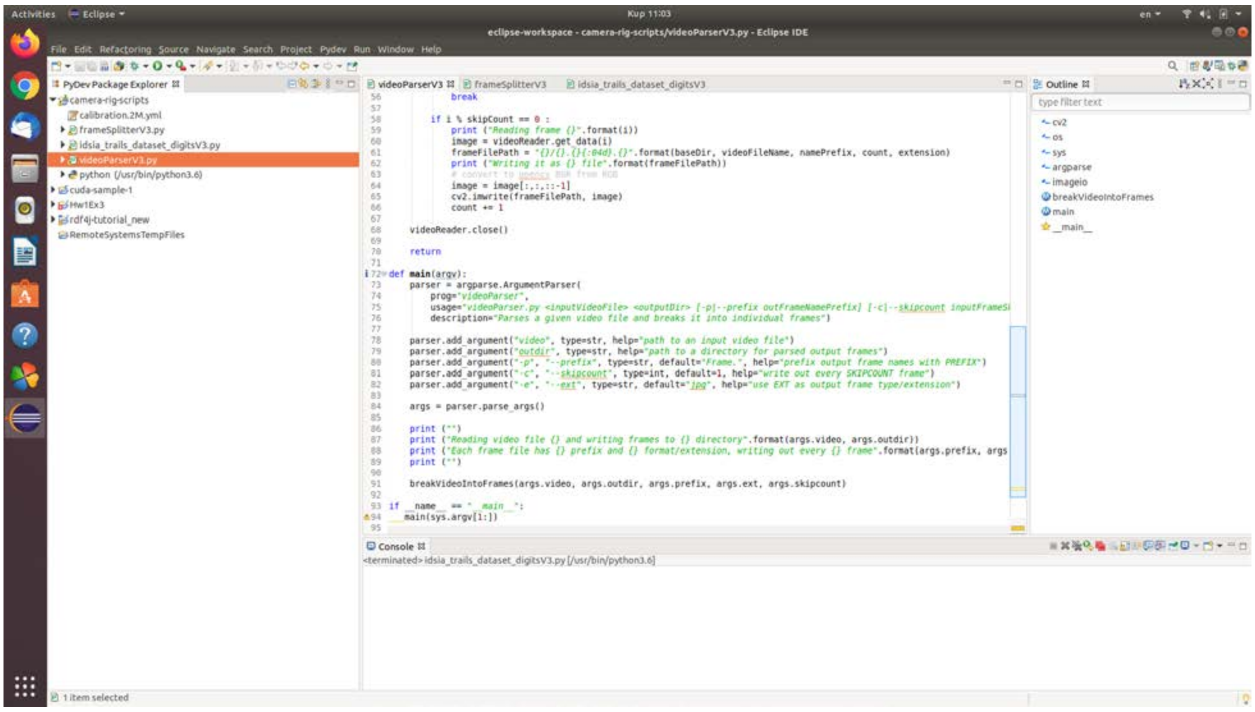
As we needed to get a view from recording cameras as close to the FOV of the drone camera, we decided to divide each 120^0 FOV to three separate virtual 60^0 FOV. So in total there would be 9 simultaneous viewpoints, but only the 3 central views were mandatory for use in order to train the lateral offset layer.

For undistorting image frames and then split them into three virtual views, we used the python script «frameSplitterV3.py». Each 120^0 frame is split to three 60^0 virtual views that have orientation 25^0 left, straight and 25^0 right. The three central generated virtual views are used as input for training the DNN's lateral offset layers.

Image 17: Indicative frames from (a) left, (b) center, and (c) right central virtual views of our dataset

II.D Using Eclipse IDE and training with DIGITS

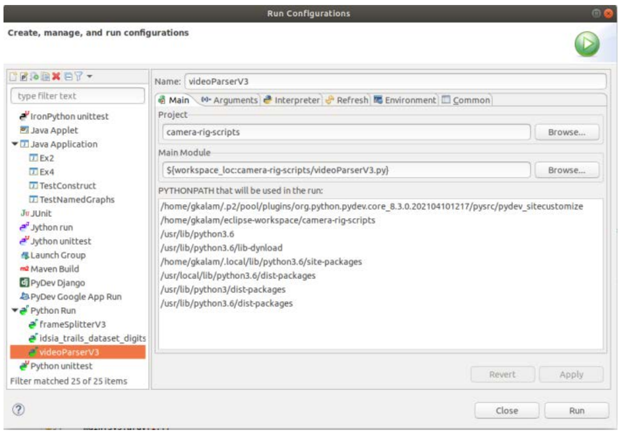
For our convenience, we used Eclipse IDE (with PyDev module) for developing and running the python scripts. In Image 18, an overview from the IDE is shown. It is the main panel where you can edit the python code. From "Run Configurations" menu, you can choose the python file for execution [Image 19]. You can also pass arguments through the "arguments" tab [Image 20].



```
            break

        if i % skipCount == 0 :
            print ("Reading frame {}".format(i))
            image = videoReader.get_data(i)
            frameFilePath = "{}/{}.{}{:04d}.{}".format(baseDir, videoFileName, namePrefix, count, extension)
            print ("Writing it as {} file".format(frameFilePath))
            # convert to opencv BGR from RGB
            image = image[:,:,::-1]
            cv2.imwrite(frameFilePath, image)
            count += 1

    videoReader.close()

    return

def main(argv):
    parser = argparse.ArgumentParser(
        prog="videoParser",
        usage="videoParser.py <inputVideoFile> <outputDir> [-p|--prefix outFrameNamePrefix] [-c|--skipcount inputFrameSk
        description="Parses a given video file and breaks it into individual frames")

    parser.add_argument("video", type=str, help="path to an input video file")
    parser.add_argument("outdir", type=str, help="path to a directory for parsed output frames")
    parser.add_argument("-p", "--prefix", type=str, default="Frame.", help="prefix output frame names with PREFIX")
    parser.add_argument("-c", "--skipcount", type=int, default=1, help="write out every SKIPCOUNT frame")
    parser.add_argument("-e", "--ext", type=str, default="jpg", help="use EXT as output frame type/extension")

    args = parser.parse_args()

    print ("")
    print ("Reading video file {} and writing frames to {} directory".format(args.video, args.outdir))
    print ("Each frame file has {} prefix and {} format/extension, writing out every {} frame".format(args.prefix, args
    print ("")

    breakVideoIntoFrames(args.video, args.outdir, args.prefix, args.ext, args.skipcount)

if __name__ == "__main__":
    main(sys.argv[1:])
```

```
<terminated> idsia_trails_dataset_digitsV3.py [/usr/bin/python3.6]
```

Image 18: Eclipse IDE – Overview from PyDev perspective



Name: videoParserV3

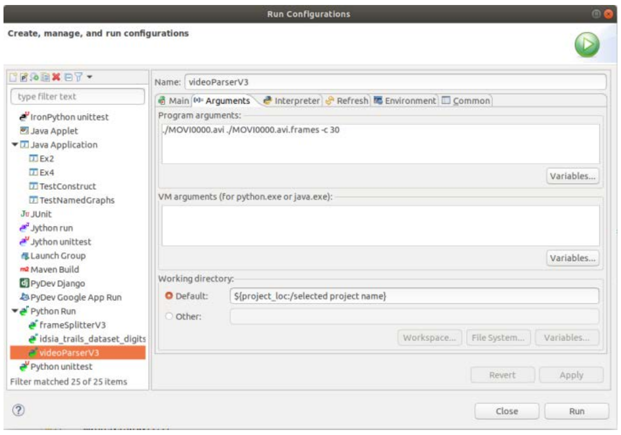
Project: camera-rig-scripts

Main Module: ${workspace_loc:camera-rig-scripts/videoParserV3.py}

PYTHONPATH that will be used in the run:

```
/home/gkalam/.p2/pool/plugins/org.python.pydev.core_8.3.0.202104101217/pysrc/pydev_sitecustomize
/home/gkalam/eclipse-workspace/camera-rig-scripts
/usr/lib/python3.6
/usr/lib/python3.6/lib-dynload
/home/gkalam/.local/lib/python3.6/site-packages
/usr/local/lib/python3.6/dist-packages
/usr/lib/python3/dist-packages
/usr/lib/python3.6/dist-packages
```

Image 19: Eclipse IDE – Run Configurations main tab



Name: videoParserV3

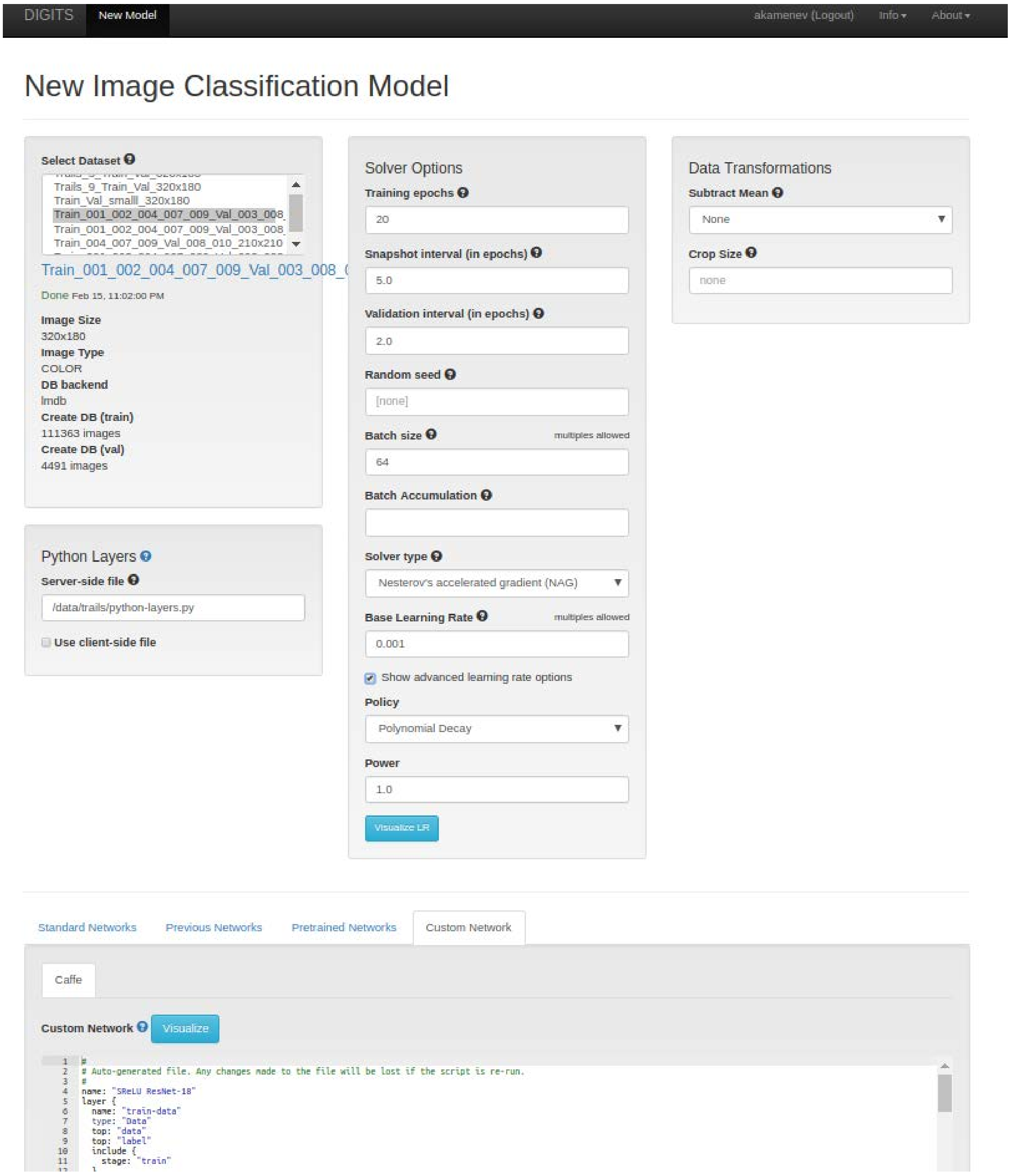
Program arguments:

```
./MOVI0000.avi ./MOVI0000.avi.frames -c 30
```

VM arguments (for python.exe or java.exe):

Working directory:
[✓ Default:] ${project_loc:/selected project name}
Other:

Image 20: Eclipse IDE – Run Configurations Arguments tab



New Image Classification Model

Select Dataset
Trails_9_Train_Val_320x180
Train_Val_smalll_320x180
Train_001_002_004_007_009_Val_003_008_
Train_001_002_004_007_009_Val_003_008_
Train_004_007_009_Val_008_010_210x210

Train_001_002_004_007_009_Val_003_008_(

Done Feb 15, 11:02:00 PM

Image Size
320x180
Image Type
COLOR
DB backend
lmdb
Create DB (train)
111363 images
Create DB (val)
4491 images

Python Layers
Server-side file
/data/trails/python-layers.py
☐ **Use client-side file**

Solver Options
Training epochs
20
Snapshot interval (in epochs)
5.0
Validation interval (in epochs)
2.0
Random seed
[none]
Batch size multiples allowed
64
Batch Accumulation

Solver type
Nesterov's accelerated gradient (NAG)
Base Learning Rate multiples allowed
0.001
☑ Show advanced learning rate options
Policy
Polynomial Decay
Power
1.0
Visualize LR

Data Transformations
Subtract Mean
None
Crop Size
none

Standard Networks | Previous Networks | Pretrained Networks | Custom Network

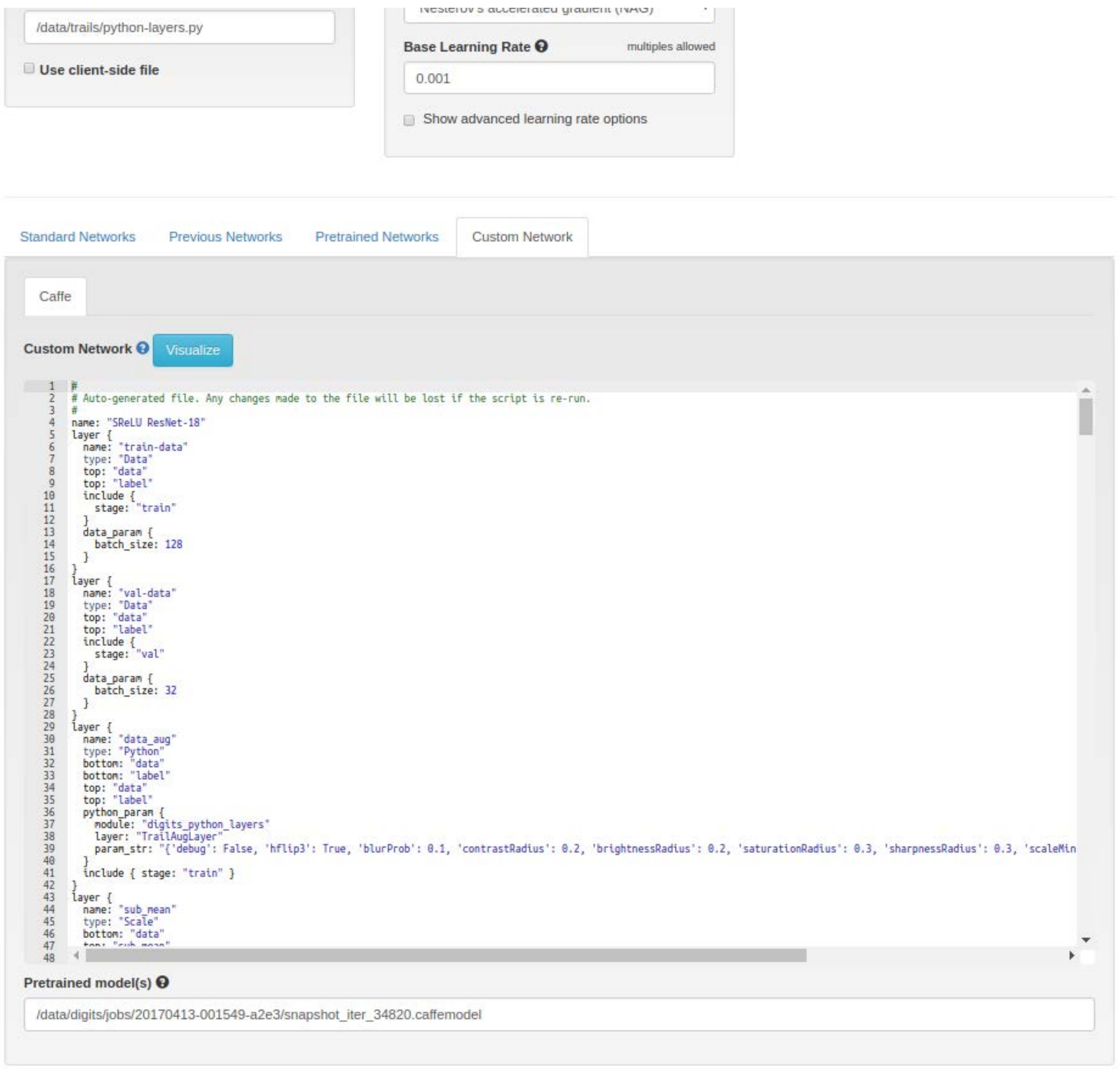
Caffe

Custom Network Visualize

```
#
# Auto-generated file. Any changes made to the file will be lost if the script is re-run.
#
name: "SReLU ResNet-18"
layer {
  name: "train-data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    stage: "train"
```

Image 21: Training Head Direction (HD) through DIGITS UI

As shown in Image 21, we used DIGITS UI for training of the Head Direction (HD). First we created a new Image Classification Model. Then we selected the IDSIA dataset. We gave all hyper-parameters as shown in Table 1. Then we provided the full path to Python layers file located at "autonomous_drones/models/nets/python-layers.py". On Custom Network tab we pasted network definition from "autonomous_drones/models/nets/ResNet/srelu-resnet-18.prototxt". Then we pushed the "Create" button and wait until the training was finished.



/data/trails/python-layers.py

☐ Use client-side file

Nesterov's accelerated gradient (NAG)

Base Learning Rate — multiples allowed

0.001

☐ Show advanced learning rate options

Custom Network

```
#
# Auto-generated file. Any changes made to the file will be lost if the script is re-run.
#
name: "SReLU ResNet-18"
layer {
  name: "train-data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    stage: "train"
  }
  data_param {
    batch_size: 128
  }
}
layer {
  name: "val-data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    stage: "val"
  }
  data_param {
    batch_size: 32
  }
}
layer {
  name: "data_aug"
  type: "Python"
  bottom: "data"
  bottom: "label"
  top: "data"
  top: "label"
  python_param {
    module: "digits_python_layers"
    layer: "TrailAugLayer"
    param_str: "{'debug': False, 'hflip3': True, 'blurProb': 0.1, 'contrastRadius': 0.2, 'brightnessRadius': 0.2, 'saturationRadius': 0.3, 'sharpnessRadius': 0.3, 'scaleMin
  }
  include { stage: "train" }
}
layer {
  name: "sub_mean"
  type: "Scale"
  bottom: "data"
  top: "sub_mean"
```

Pretrained model(s)

/data/digits/jobs/20170413-001549-a2e3/snapshot_iter_34820.caffemodel

Image 22: Training Lateral Offset (LO) through DIGITS UI

For training the Lateral Offset (LO), we followed the above procedure with minor changes. We used “TrailNet_SResNet-18.prototxt” model definition file. Then we provided full path to pretrained orientation model in Pretrained model(s) field on Custom Network tab as shown in Image 22.

APPENDIX III: Docker Support

Linux4Tegra comes with a preinstalled set of system components, one of which is the `nvidia-container`. NVIDIA offers a repository for building and running docker jetson-containers[12].

For our experiments, we decided to work with ROS Noetic targeted for Ubuntu 20.04 with End-Of-Life in May 2025. For our Jetpack (version 32.5) we choose from Dusty'sNV DockerHub the ROS Noetic[13] with tag `ros:noetic-ros-base-l4t-r32.5.0`.

In path `~pcomp\nvidia-tutorials\` you can find `run_rosnoetic_docker.sh` which fires up a docker container for ros-noetic image. ROS nodes can be started inside the container's console. Files and documents can be found in our github repo [Appendix I], in folder `docker-run-scripts`.

```
$ sudo service docker start
$ service --status-all | grep docker
$ docker version
$ docker info
$ sudo service docker restart

#Clear dangling images
$ docker system prune --all --volumes --force
#freed up disk space from volumes
$ docker volume rm $(docker volume ls -qf dangling=true)

#removing all docker logs files from my containers
$ find /var/lib/docker/containers/ -type f -name “*.log” -delete
$ sudo sh -c "truncate -s 0 /var/lib/docker/containers/*/*-json.log"
#restart the docker containers
$ docker-compose down && docker-compose up -d
#clean up the builder cache
$ docker builder prune –all

#docker basic commands
$ docker image ls
$ docker container ls
$ docker volume ls      # mounted paths

# docker remove image
$ docker image rm <docker_image_id>
$ docker image rm c74a2c7e3669
```

Figure 9: Docker commands for managing images, containers and other functionalities

Another repository containing Docker Images is the NVIDIA GPU CLOUD[14] (NGC). You can navigate through CATALOG | CONTAINERS and use as query parameter “l4t”. Some of the containers used are:

[12] https://github.com/dusty-nv/jetson-containers

[13] https://hub.docker.com/r/dustynv/ros

[14]https://ngc.nvidia.com/catalog/containers/?orderBy=scoreDESC&pageNumber=0&query=l4t&quickFilter=&filters=

DLI Getting Started with AI on Jetson Nano[15]: A series of introductory lessons on how to use basic functionalities and features on Jetson Nano and TX2.

NVIDIA L4T Base[16]: The Linux4Tegra OS inside a Docker image.

NVIDIA L4T ML[17]: The `l4t-ml` docker image contains TensorFlow, PyTorch, JupyterLab, and other popular ML and data science frameworks such as scikit-learn, scipy, and Pandas pre-installed in a Python 3.6 environment.

Libraries included in JetPack 4.5 (L4T R32.5.0)

```
    l4t-ml:r32.5.0-py3
        TensorFlow 1.15
        PyTorch v1.7.0
        torchvision v0.8.0
        torchaudio v0.7.0
        onnx 1.8.0
        CuPy 8.0.0
        numpy 1.19.4
        numba 0.52.0
        OpenCV 4.1.1
        pandas 1.1.5
        scipy 1.5.4
        scikit-learn 0.23.2
        JupyterLab 2.2.9
```

NVIDIA Deep Learning GPU Training System (DIGITS)[18]: DIGITS is a web app for training deep learning models, and currently supports the TensorFlow framework. DIGITS can be used to rapidly train highly accurate deep neural network (DNNs) for image classification, segmentation, object detection tasks, and more. For the needs of this project we use the caffe framework with tag “*20.03-caffe-py3*”.

[15] https://ngc.nvidia.com/catalog/containers/nvidia:dli:dli-nano-ai

[16] https://ngc.nvidia.com/catalog/containers/nvidia:l4t-base

[17] https://ngc.nvidia.com/catalog/containers/nvidia:l4t-ml

[18] https://ngc.nvidia.com/catalog/containers/nvidia:digits

REFERENCES

[1] Giusti, A., Guzzi, J., Ciresan, D.C., He, F.-L., Rodrıguez, J.P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Di Caro, G., et al.: A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robot. Autom. Lett. 1(2)*, 661–667 (2015)

[2] Smolyanskiy, N., Kamenev, A., Smith, J., Birchfield, S.: Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4241–4247 (2017)

[3] Rasmussen, C., Lu, Y., Kocamaz, M.: Appearance contrast for fast, robust trail-following. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3505–3512 (2009)

[4] Santana, P., Correia, L., Mendon˙ca, R., Alves, N., Barata, J.: *Tracking natural trails with swarm-based visual saliency. J. Field Robot. 30(1)*, 64–86 (2013)

[5] Levin, A., Weiss, Y.: *Learning to combine bottom-up and top-down segmentation. Int. J. Comput. Vis. 81(1)*, 105–118 (2009)

[6] *IDSIA forest trail dataset*, 2015, http://bit.ly/perceivingtrails [Accessed on 15/09/2021]

[7] C. Rasmussen, Y. Lu, and M. Kocamaz, *"A trail-following robot which uses appearance and structural cues," in Field and Service Robotics*, pp. 265–279, Springer, 2014.

[8] Maciel-Pearson, B.G., Carbonneau, P., Breckon, T.P.: Extending Deep Neural Network Trail Navigation for Unmanned Aerial Vehicle Operation within the Forest Canopy. In: *Annual Conference Towards Autonomous Robotic Systems*, pp. 147–158. Springer, Cham (2018)

[9] Zhilenkov, A.A., Epifantsev, I.R.: System of Autonomous Navigation of the Drone in Difficult Conditions of the Forest Trails. In: *IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConrus)*. IEEE (2018)

[10] Palossi, D., Loquercio, A., Conti, F., Flamand, E., Scaramuzza, D., Benini, L.: Ultra Low Power Deep-Learning-Powered Autonomous Nano Drones. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2018)* (2018)

[11] M. Bojarski et al. End to End Learning for Self-Driving Cars. *arXiv preprint*, arXiv:1604.07316v1, 2016.

[12] H. Alvarez, L. Paz, J. Sturm, and D. Cremers. Collision avoidance for quadrotors with a monocular camera. In *International Symposium on Experimental Robotics (ISER)*, pp. 195–209, Springer, 2016

[13] A. Bry, A. Bachrach, and N. Roy. State estimation for aggressive flight in GPS-denied environments using onboard sensing. In *International Conference on Robotics and Automation (ICRA)*, 2012.

[14] F. Fraundorfer, H. Lionel, D. Honegger, G. Lee, L. Meier, P. Tanskanen, and M. Pollefeys. Vision-based autonomous mapping and exploration using a quadrotor MAV. In *International Conference on Intelligent Robots and Systems (IROS)*, Nov. 2012.

[15] D. Scaramuzza et al. Vision-controlled micro flying robots: From system design to autonomous navigation and mapping in GPS-denied environments. *IEEE Robot. Auton. Mag.*, 21(3):26–40, Sep. 2014.

[16] Mori, T., Scherer, S.: First results in detecting and avoiding frontal obstacles from a monocular camera for micro unmanned aerial vehicles. In: *2013 IEEE International Conference on Robotics and Automation*, pp. 1750–1757 (2013)

[17] Al-Kaff, A., Meng, Q., Mart´ın, D., de la Escalera, A., Armingol, J.M.: Monocular vision-based obstacle detection/avoidance for unmanned aerial vehicles. In: *IEEE Intelligent Vehicles Symposium (IV)*, pp. 92–97 (2016)

[18] Drews, P., de Bem, R., de Melo, A.: Analyzing and exploring feature detectors in images. In: *2011 9th IEEE International Conference on Industrial Informatics*, pp. 305–310. IEEE (2011)

[19] Smolyanskiy, N., Kamenev, A., Birchfield, S.: On the Importance of Stereo for Accurate Depth Estimation: An Efficient Semi-Supervised Deep Neural Network Approach. *arXiv preprint*, arXiv:1803.09719, 2018

[20] Pixhawk, https://pixhawk.org/ [Accessed on 15/09/2021]

[21] N. Deligiannakis, "A Mixed Reality Dashboard based on a distributed data streaming architecture", master's thesis, Dept. of Informatics and Telecommunications, National and Kapodistrian Univ. of Athens, 2020.

[22] V. Iliopoulos, F. Theodoulou, "Framework for autonomous navigation through MS HoloLenses", bachelor's thesis, Dept. of Informatics and Telecommunications, National and Kapodistrian Univ. of Athens, 2021.

[23] J. Engel, V. Koltun, D. Cremers. Direct sparse odometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[25] Girshick, Ross, Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Sergio Guadarrame, and Trevor Darrell. 2014. "Caffe: Convolutional Architecture for Fast Feature

Embedding." In *Proceedings of the 22nd ACM International Conference on Multimedia*: 675 – 678. https://arxiv.org/abs/1408.5093.

[26] Girshick, Ross. 2015. "Fast R-CNN." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*: 1440–1448. http://doi.org/10.1109/ICCV.2015.169.

[27] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2015. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." In Proceeding of 2016 IEEE Transactions on Pattern Analysis and Machine Intelligence: 1137–1149. http://doi.org/10.1109/TPAMI.2016.2577031.

[28] Xu, Joyce. 2017. "Deep Learning for Object Detection: A Comprehensive Review." Towards Data Science. September 11, 2017. https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9.

[29] Gandhi, Rohith. 2017. "Support Vector Machine — Introduction to Machine Learning Algorithms." Towards Data Science. June 7, 2018. https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[30] Redmon, Joseph, Santosh Diwala, Ross Girshick, and Ali Farhadi. 2016. "You Only Look Once: Unified, Real-Time Object Detection." In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*: 779 – 788. https://doi.org/10.1109/CVPR.2016.91.

[31] J. Redmon and A. Farhadi. YOLO9000: Better, faster, stronger. *arXiv preprint*, arXiv:1612.08242, 2016.

[32] Redmon, Joseph and Farhadi, Ali, YOLOv3: An Incremental Improvement. *arXiv preprint*, arXiv: 1804.02767, 2018

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.- Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[34] Hossain, S.; Lee, D.-j. Deep Learning-Based Real-Time Multiple-Object Detection and Tracking from Aerial Imagery via a Flying Robot with GPU-Based Embedded Devices. *Sensors 2019*, 19, 3371. https://doi.org/10.3390/s19153371

[35] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." arXiv preprint arXiv:2004.10934 (2020).

[36] Wang, Chien-Yao, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. "Scaled-YOLOv4: Scaling Cross Stage Partial Network." arXiv preprint arXiv:2011.08036 (2020).

[37] JK Jung's blog, TensorRT YOLOv4, Jul 2020, https://jkjung-avt.github.io/tensorrt-yolov4/ [Accessed on 15/09/2021]

[38] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll´ar, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 740–755, 2014.

[39] JK Jung's GitHub, "TensorRT demos: TensorRT MODNet, YOLOv4, YOLOv3, SSD, MTCNN, and GoogLeNet", https://github.com/jkjung-avt/tensorrt_demos [Accessed on 15/09/2021]

[40] M. Sandler et al., "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[41] Verucchi, Micaela and Brilli, Gianluca and Sapienza, Davide and Verasani, Mattia and Arena, Marco and Gatti, Francesco and Capotondi, Alessandro and Cavicchioli, Roberto and Bertogna, Marko and Solieri, Marco, A Systematic Assessment of Embedded Neural Networks for Object Detection, in Proceedings of the *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020, vol1, pages 937—944

[42] DJI's Guidance System, https://www.dji.com/gr/guidance [Accessed on 15/09/2021]

[43] DJI's Matrice 100 UAV Specifications, https://www.dji.com/gr/matrice100 [Accessed on 15/09/2021]

[44] NVIDIA Jetson TX2 Developer Kit Specifications, https://developer.nvidia.com/embedded/jetson-tx2-developer-kit [Accessed on 15/09/2021]

[45] Omnivision OV5693 CSI Camera Module Technical Specifications, https://www.ovt.com/sensors/OV5693 [Accessed on 15/09/2021]

[46] Auvidea's J120A-IMU/MCU carrier board for Jetson TX1/TX2, https://auvidea.eu/product/70719/ [Accessed on 15/09/2021]

[47] DJI's ZenMUSE Z3 Camera, https://www.dji.com/gr/zenmuse-z3 [Accessed on 15/09/2021]

[48] Youtube Video, Highbanks Virtual Trail Run, https://youtu.be/K_K_EHDaSew [Accessed on 15/09/2021]

[49] DJI Onboard SDK Documentation Online, https://developer.dji.com/onboard-sdk [Accessed on 15/09/2021]

[50] A. Martinez and E. Fernández, *Learning ROS for Robotics Programming*. Packt Publishing, 2013

[51] Cookson, A. U., Landergan, M., & O'Neil, S. T. (2019).Wolfgang: An Autonomous Mobile Robot for Outdoor Navigation. Retrieved from https://digitalcommons.wpi.edu/mqp-all/7036

[52] "About ROS," Open Source Robotics Foundation. [Online]. Available: http://www.ros.org/about-ros/