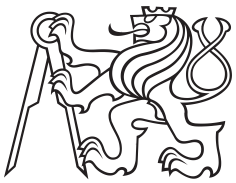
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Drone detection using neural networks from combined RGB camera and LiDAR data

Adam Škuta

Supervisor: Matouš Vrba
Supervisor–specialist: Martin Saska
Field of study: Mathematical Engineering
Subfield: Mathematical Modelling
February 2017

Acknowledgements

Děkuji ČVUT, že mi je tak dobrou *alma mater*.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 10. February 2017

Abstract

Keywords: word, key

Supervisor: Matouš Vrba
Ústav X,
Uliční 5,
Praha 99

Abstrakt

Klíčová slova: slovo, klíč

Překlad názvu: Moje bakalářka se strašně, ale hrozně dlouhým předlouhým názvem — Cesta do tajů kdovíčeho

Contents

Figures

Tables

ctuthesis t1606152353

Chapter 1

Introduction

In this thesis a drone detection using neural networks from combined RGB camera and LiDAR data is studied. With the recent development of drone technology, drones have become more readily available to public and could only be expected to rise in popularity in the future. Drone detection is an important problem to tackle when it comes to tasks such as interception of uncooperative drones [VHS19] or localization of drones in swarm [KNF+14] [TBH+19]. Furthermore Light detection and ranging (LiDAR) sensors mounted on drones have been utilized [LHK16] and provide a viable sensor option. Therefore fusing camera and LiDAR sensor could be used for more accurate localization.

The goal of this thesis was to examine whether a usage of LiDAR data coupled with RGB images from camera is useful for the localization of drones in contrast to the usage of image data alone.The LiDAR and RGB camera were mounted on top of the observer drone, which took pictures and point-clouds of the target drone. All the measurements were taken inside a virtual environment, with a realistic drone and sensor simulation. The dataset was then processed and used as the input for training and testing a Convolutional neural network for the object detection. The preprocessing was as following:

- Coordinate transformation for the non-matching coordinate systems,
- Projection of 3d points into a 2d image,
- Utilizing different processing methods on sparse LiDAR point cloud in order to make it more dense,
- Fusing RGB images and LiDAR data into RGBD images.

The output metrics were then compared with the RGB trained convolutional neural network metrics.

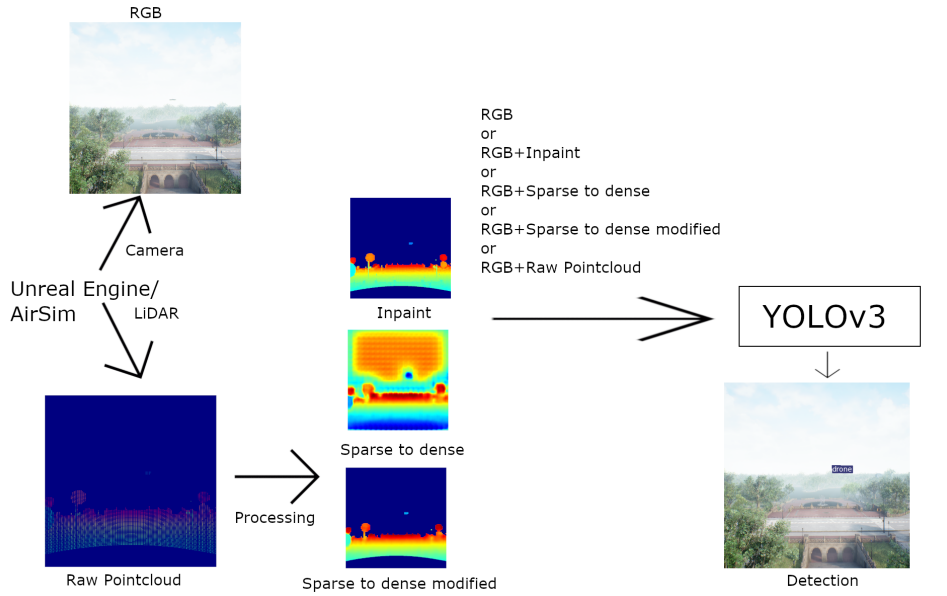


Figure 1.1: Schema of the process.

1.1 Related Work

Solving the problem of drone detection has been tackled in different ways. The main difference between them is the choice of sensor or sensors utilized.

Chapter 2

Methodolology

In this chapter various methods used in this thesis are discussed. A video game engine Unreal Engine[1]paired with plugin AirSim[2] was used for simulating real-life environments and for generating the dataset. An image inpainting technique was used as an option for transforming sparse LiDAR pointcloud into a dense one. A convolutional neural network was used in two problems in the thesis. First one was on a sparse LiDAR pointcloud generating more points and therefore making it more dense. Second one was used for object, in this case drone, detection using RGB and RGBD data as the input.

2.1 Sparse to dense

Sparse to dense is a Convolutional neural network written in PyTorch. [MK18] It predicts the depth measurements from sparse depth dataset. The size of the network is modifiable and can be chosen as training parameters. For the encoding and therefore input layers a ResNet-50 or ResNet-18 can be chosen, depending on the size of the input image for memory constraints. The decoding layers consist of 4 upsampling layers and a deconvolutional layer with either stride 2 or 3 or uprojection layer or upconvolutional layer as a choice for training. The default loss function is least absolute deviations

[1] https://www.unrealengine.com
[2] https://microsoft.github.io/AirSim/

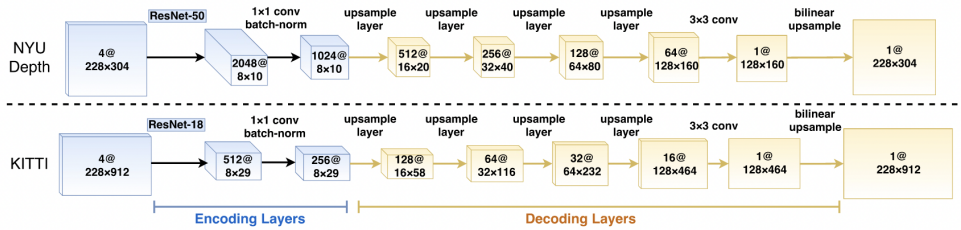


Figure 2.1: Example of different network architectures available. Taken from [MK18].

also known as L_1 error:

$$L_1 = \sum_{i=1}^{n} |y_{true} - y_{predicted}|, \tag{2.1}$$

where:

- n is batch size,
- y_{true} are real depth values,
- $y_{predicted}$ are predicted depth values.

The input to the network are RGBD images and the output is a depth map with the same dimensions as input. The depth input D is sampled from the ground truth depth map D^* with the following formula:

$$D(i,j) = \begin{cases} D^*(i,j), & \text{with probability } p, \\ 0, & \text{otherwise,} \end{cases} \tag{2.2}$$

where:

- i, j are coordinates of the input image,
- $p = \frac{m}{n}$, where m number of depth samples to be chosen at the start of training and n is the total amount of available depth samples.

During training several input data augmentations take place. These augmentations include:

- Scaling the input image by a random number $s \in [1, 1.5]$,

- Rotating the input image by a random degree $r \in [-5, 5]$,
- Scaling the brightness, contrast and saturation of the RGB component of the image by a random number $k \in [0.6, 1.4]$,
- Normalizing the RGB component of the image,
- Flipping the image horizontally with a 50% chance.

The output of the network is a dense depth image with the dimensions of the input. Every pixel contains predicted depth measurement in meters. The output of the Sparse to Dense network will be used for further training later.

2.2 YOLOv3

citations

You only look once (YOLO) is a convolutional neural network model used mainly for object detection and recognition[RDGF16]. The main advantage is its simplicity in comparison to similar convolutional neural networks, resulting in faster detection speeds. It belongs to the state of the art convolutional neural networks for object detection and recognition. The version used in this work is the third version YOLOv3[RF18]. The backbone called Darknet53 consists of 53 convolutional layers. The original detector consists of 3 detection layers each responsible for detecting objects of various sizes. At each detection

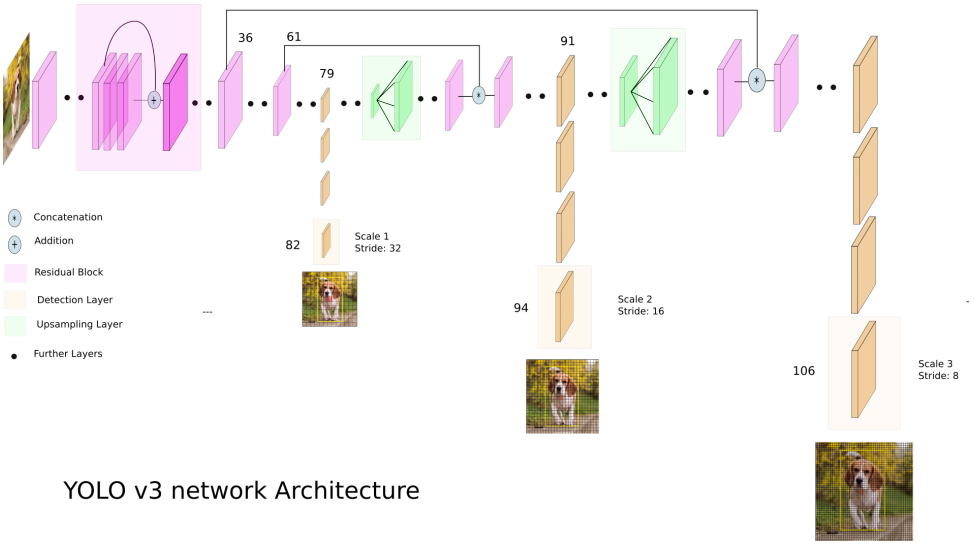


Figure 2.2: YOLOv3 network architecture. Taken from [Kat18]

layer, the image is divided into multiple grid cells, where each grid cell detects

three bounding boxes. YOLOv3 takes n-channel images as the input. Each of the detection layers outputs three bounding boxes for each of the cell. The content of one bounding box is as following:

- t_x, t_y, t_w, t_h are bounding box co-ordinates,
- p_O is objectness score,
- p_c is class score for each class in the dataset.

The bounding box t_x and t_y coordinates are relative to the upper-left corner of its respective cell, while t_w and t_h are relative to one of the anchors. Anchors are pre-defined default bounding box sizes and can be modified before training. To transform these bounding box coordinates to be relative to the image following transformation is applied:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x, \\ b_y &= \sigma(t_y) + c_y, \\ b_w &= p_w e^{t_w}, \\ b_h &= p_w e^{t_h}, \end{aligned} \tag{2.3}$$

where:

- $\sigma(x)$ is a sigmoid function,
- c_x, c_y are grid cells offsets from the top left corner of the image,
- p_w, p_h are anchors width and height respectively,
- b_x, b_y, b_w, b_h are bounding box coordinates relative to the image size.

The loss function used during the training is sum squared error loss or L_2 error described as follows:

$$L_2 = \sum_{i=1}^{n} (\hat{\mathbf{t}} - \mathbf{t})^2, \tag{2.4}$$

where:

- n is batch size,
- $\hat{\mathbf{t}}$ is vector of predicted bounding box co-ordinates,
- $\mathbf{t}$ is vector of ground truth bounding box coordinates which can be obtained by inverting transformation in 2.3.

2.3 Image inpainting method based on the Fast Marching Method

Image inpainting is a method used for reconstructing missing values in the image. A one such method based on [Tel04] called Image inpainting method based in the Fast Marching Method is presented in this section. A grayscale image is used for simplification purposes.

The grayscale value of a pixel to be inpainted is determined by the known neighboring pixel values. To compute grayscale value from one close pixel a following formula is used:

$$I_q(p) = I(q) + \nabla I(q)(p - q), \tag{2.5}$$

where:

- q are pixel co-ordinates with known grayscale value,
- p are pixel co-ordinates with unknown grayscale value,
- $I(x)$ is a grayscale value at pixel co-ordinates x,
- $\nabla I(x)$ is a gradient value at pixel co-ordinates x.

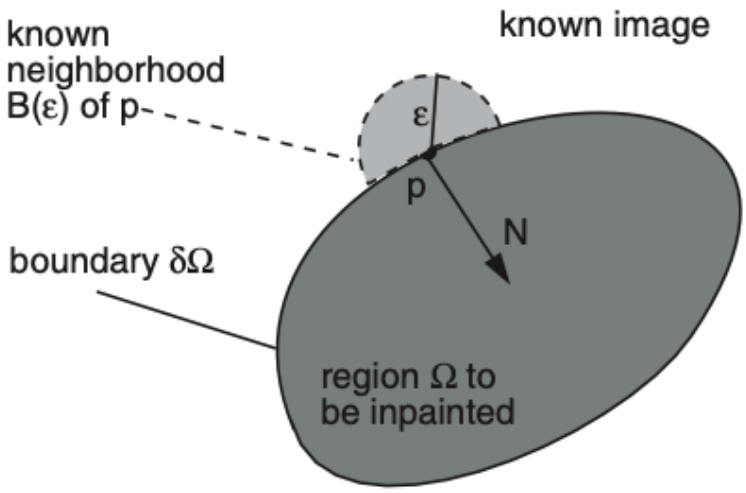


Figure 2.3: Inpainting principle. Image from: [Tel04]

To get a final value for the unknown pixel the Equation 2.5 is applied on all known pixels in a specified region $B_\varepsilon(p)$ from the unknown pixel. The

function is as following:

$$I(p) = \frac{\sum_{q \in B_\varepsilon(p)} w(p,q) I_q(p)}{\sum_{q \in B_\varepsilon(p)} w(p,q)}, \tag{2.6}$$

where $w(p,q)$ is a weighting function designed for propagating sharpness of the image and is obtained as the product of the following equations:

$$\begin{aligned} dir(p,q) &= \frac{p-q}{||p-q||} N(p), \\ dst(p,q) &= \frac{1}{||p-q||^2}, \\ lev(p,q) &= \frac{1}{1+|T(p)-T(q)|}, \end{aligned} \tag{2.7}$$

where,

- $N(x)$ is a normal direction of the boundary to be inpainted at pixel co-ordinates x,
- $T(x)$ is distance of pixel x to the inpainting boundary.

The equation 2.6 is iteratively applied to all pixels on the inpainting boundary, and advances inside the region to be inpainted until the whole region has been filled. This is implemented via Fast Marching Method algorithm.

Figure 2.4: Example of the inpainting technique. Image from: [Tel04]

2.4 Coordinate systems

In order to correctly label the data for training, a position of the target drone in relation to the sensor mounted on the observer drone is required. The AirSim API returns the position of each drone in respect to their starting points. The starting point is a point where the drone spawns in the map. Therefore a transformation from the starting point of the target drone to the camera mounted on the observer is required. This transformation is written as follows:

$$\mathbf{T} = \mathbf{T}_o^c \mathbf{T}_{os}^o \mathbf{T}_{ts}^{os}, \tag{2.8}$$

where:

- $\mathbf{T}_{ts}^{os}$ is transformation from the starting point of the target drone to the starting point of the observer drone,
- $\mathbf{T}_{os}^{o}$ is transformation from the starting point of the to the body of the first drone,
- $\mathbf{T}_{o}^{c}$ is transformation from the body of the first drone to the cameras coordinate system.

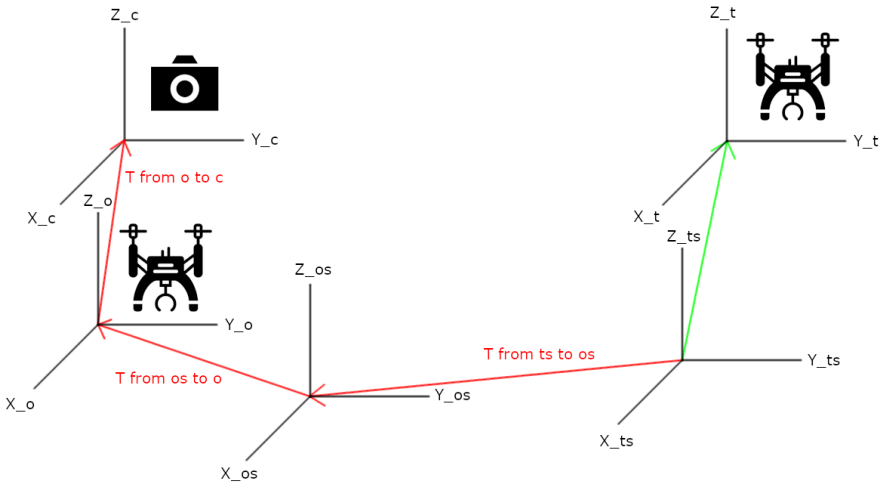


Figure 2.5: Visualization of transformation.

Transformation matrix $\mathbf{T}$ is generally be described as follows:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0}^T & 1 \end{bmatrix}, \tag{2.9}$$

where:

- **R** is a 3x3 rotation matrix,
- **p** is a 3x1 translation column vector,
- $\mathbf{0}^T$ is a 1x3 row vector of zeros.

2.5 Camera Model

For the creation of the bounding boxes used for training and for processing raw data from LiDAR sensor a transformation from coordinate system of the camera to the pixel values of the image needs to be defined. For this task a pinhole camera model is used.

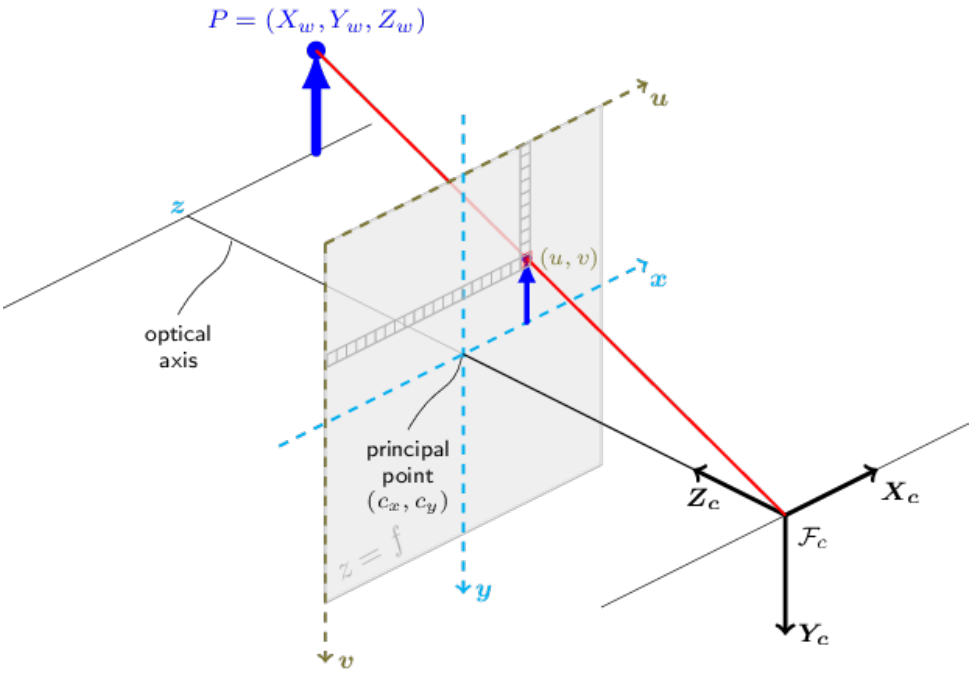


Figure 2.6: Pinhole camera model. Image from [ope].

The transformation is then defined as follows:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f\frac{X_c}{Z_c} + c_x \\ f\frac{Y_c}{Z_c} + c_y \end{bmatrix}, \tag{2.10}$$

where:

- u and v are pixel coordinate values on the image,

- X_c,Y_c,Z_c are coordinate values of a point in the coordinate system of the camera,
- f is focal length of the camera,
- c_x,c_y are offsets on the image plane.

In order to transform the point from world coordinates X_w, Y_w, Z_w to the coordinate system of the camera a following transformation needs to be applied:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.11}$$

Pinhole camera model is only idealization of a real life camera and no lens distortion needs to be considered. The AirSim simulator simulates pinhole camera so no other processing needs to be done.

2.6 Dataset

The dataset for this work can be generated in two ways. The first is real-life drone shots mixed with point clouds from LiDAR mounted on top of a drone. The second is generating a dataset using a realistic virtual environment where a drone, camera and LiDAR are being emulated very close to their real-life counterparts. An advantage to this approach is that a great variety of environments can be chosen a lot of them often inaccessible otherwise (power plant, airport, snowy mountains out of season etc.). Therefore this approach was chosen for the task.

2.6.1 Unreal Engine

Unreal Engine is a software tool used for creating realistic 3d environments, most often used as a video game engine. It is written in C++ and open-source supporting a variety of pre-built environments and assets. For this work three different environments were used for the creation of the dataset:

- City Park Environment Collection (2256 samples taken),

- Automotive Winter Scene (1813 samples taken),
- Downtown West Modular Pack(1120 samples taken).

Figure 2.7: Unreal Engine user interface.

Together 5320 pictures and labels were generated using two drones. Observer drone was equipped with RGB camera and LiDAR sensor and was responsible for taking the pictures and pointclouds from LiDAR. The target drone was used as a model for drone detection.

2.6.2 AirSim

Open-source plugin for Unreal Engine called AirSim was used for the generation of the dataset. It simulates realistic flight motions of drones as well as seven types of sensors, including RGB camera and LiDAR used for this task. AirSim supports both a C++ API as well as Python API, latter which was used for controlling the motion and capturing the dataset. Location of the second drone was generated through API call, which produces a location of the drone in global coordinate system of the map, which is later transformed to the local coordinates of the first drone carrying the LiDAR and RGB sensors using Equation 2.10. The capturing drone traveled on each map on a 3D cube grid.

(a) : Snow Environment.

(b) : City Environment.

(c) : Park Environment.

Figure 2.8: Sample photos from each environment.

2.7 Training

For the training purposes 5320 samples were taken using AirSim simulator. Each sample consists of:

- 640x640 RGB image from the camera,
- 640x640 sparse depth image from the LiDAR sensor,
- Label file containing ground truth bounding boxes co-ordinates.

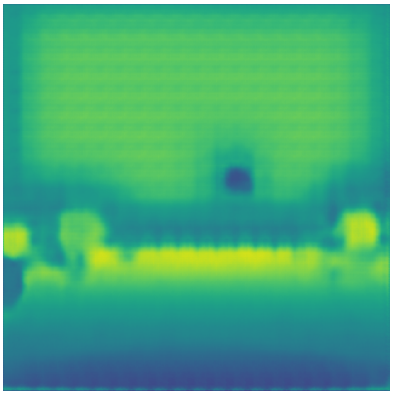
This dataset was split into 3662 training, 407 validation, 1251 testing samples.

2.7.1 Sparse to Dense

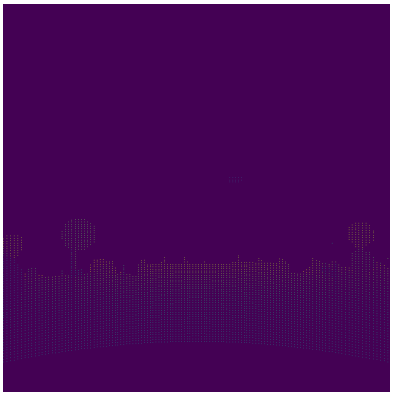
The data were trained using Sparse to Dense neural network to receive dense depth image. The training was done for 15 epochs using batch size of 8. The backbone was Resnet18 and decoder was set to Deconv3 as described in section 2.1. The network was pretrained on Kitti[3] dataset. A processing algorithm was applied to the output depth map, further filtering points that were not in vicinity of the ground truth depth points. Both filtered and unfiltered depth maps were used for further training and testing to clarify their overall impact.

[3] `http://www.cvlibs.net/datasets/kitti/`

(a) : Input image.



(b) : Output depth map.



(c) : All pointcloud points.

(d) : Selected pointcloud points.

Figure 2.9: Sparse to dense training.

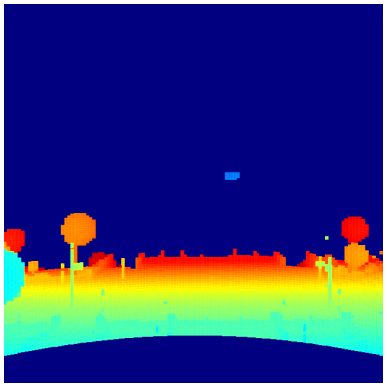
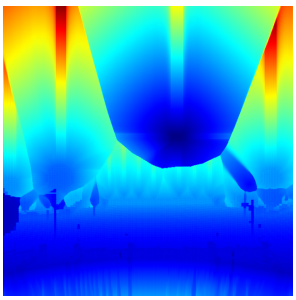


Figure 2.10: Applied filter on Sparse to dense output.

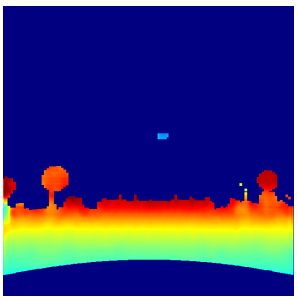
2.7.2 Image inpainting

An alternative algorithm for making sparse pointcloud into dense one was applied as well. The function is provided in OpenCV Python library. The input into the function was a sparse depth image, same as for the Sparse to

dense network except the RGB part. The algorithm introduced in 2.3 was used with the radius of 1 pixel. The results were processed with the same filtering method as for the results of Sparse to dense method. For further training only the filtered depth map was used.



(a) : Non filtered result.



(b) : Filtered result.

Figure 2.11: Inpaint results.

2.7.3 YOLOv3

Multiple different options of input to the YOLOv3 networks were applied. These include:

- Concatenating the direct output of Sparse to dense neural network with the RGB images creating a RGBD image,
- Concatenating the modified Sparse to dense output with the RGB images creating RGBD image,
- Using OpenCV library to directly impaint the sparse depth image,
- Concatenating raw LiDAR pointcloud to the RGB image
- Using only the RGB image skipping the depth entirely.

YOLOv3 PyTorch implementation was used for training[4]. Further modifications were required to be made. The original implementation of YOLOv3 supports 3-channel RGB images as inputs. For the sake of this work a RGBD input option using H5 file system was implemented. The dataset consisted of drones of various sizes ranging from very small (few pixels) to very large

[4] `https://github.com/eriklindernoren/PyTorch-YOLOv3`

closeups. Therefore 5 detection heads were implemented instead of the original 3. This ensured much higher detection confidence of smaller far away drones as well as bigger more close ones. The following parameters were used for training all inputs:

- Number of epochs was set to 120,
- Batch size was set to 128,
- Size of the input images was 416x416xn, where $n \in \{3, 4\}$,
- Learning rate was set to 0.001.

Figure 2.12: Sample YOLOv3 outputs.

Chapter 3

Results

After the training completed the different metrics on the validation dataset were compared. These metrics include:

- Precision,
- Recall,
- Intersection over Union,
- Mean Average Precision (mAP).

These metrics are given by the following formulas:

$$\begin{aligned} Precision &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}, \\ Recall &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}, \\ IoU &= \frac{\text{Area of overlap of two bounding boxes}}{\text{Area of union of two bounding boxes}}, \\ mAP &= \frac{1}{N} \sum_{i=1}^{N} AP_i. \end{aligned} \tag{3.1}$$

where:

- N is number of classes,

- AP is Average Precision, an area under the Precision Recall curve.

In this case the dataset consisted of only one class, which is drone. Therefore $mAP = AP$. For the training results validation mAP followed the training

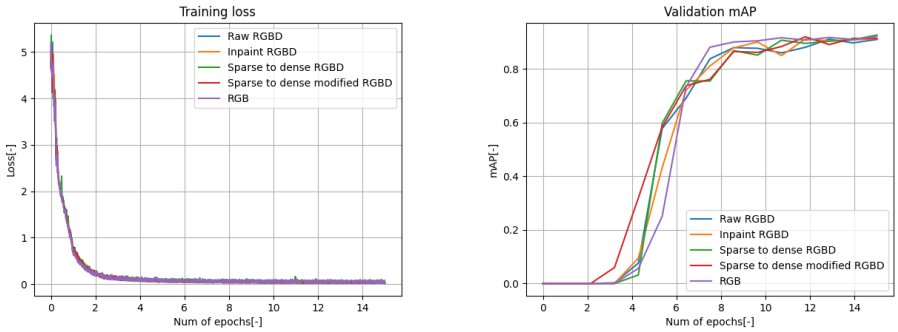


Figure 3.1: Training results

loss and started converging after around 8th epoch. It fully stopped rising after 12th epoch. Considering this the chosen weights ensured that the network does not overfit the training and validating dataset. For each input following weights were chosen:

	Raw RGBD	Inpaint RGBD	Sparse to dense RGBD	Sparse to dense modified RGBD	RGB
Weigths[epoch]	9	9	11	12	8

Table 3.1: Chosen weights.

After the weights were chosen the network was validated on the test dataset. The confidence threshold was chosen to be 0.2. The results were as following:

Results	Raw RGBD	Inpaint RGBD	Sparse to dense RGBD	Sparse to dense modified RGBD	RGB
mAP	**0.48**	0.46	0.36	0.43	0.41
Precision	0.59	0.89	**0.92**	0.60	0.79
Recall	**0.53**	0.48	0.37	0.48	0.48
IoU	0.79	**0.84**	**0.84**	0.83	0.83

Table 3.2: Testing results

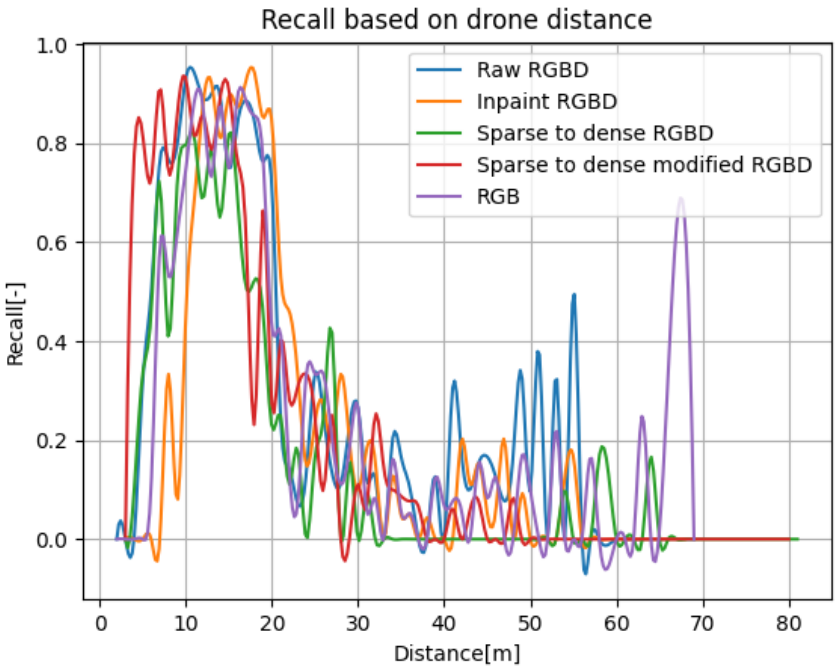


Figure 3.2: Recall based on distance of the drone.

From the results Raw RGBD offers the best improvement in terms of mAP by 7%. Every method except unmodified Sparse to dense offers some improvement in terms of mAP. In terms of precision unmodified Sparse to dense and Inpaint offer improvement over RGB by 13% and 10% respectively. Other methods namely Raw RGBD and modified Sparse to dense offer a decrease of 20% and 19% respectively. When it comes to recall not a big improvement is made. The best method is Raw RGBD with an increase of 5% in comparison to RGB. Unmodified Sparse to dense suffers a decrease of 11% in comparison to RGB, while the remaining methods are unchanged. This can be observed in Figure 3.2 where all methods perform the best in range from 4 to 19 meters reaching maximum recall of around 0.95. Modified Sparse to dense offers highest recall in around 4 meters but starts to fall after 18 meters. Raw RGBD shows a few spikes in range from around 48 to 55 meters, the highest reaching 0.5 recall. RGB shows spike at around 67 meters with recall of around 0.7. In terms of IoU all methods perform very similarly with best performing methods Inpaint RGBD and unmodified Sparse to dense showing improvement of 1%.

Speed	Raw RGBD	Inpaint RGBD	Sparse to dense RGBD	Sparse to dense modified RGBD	RGB
time[s]	0.0610	0.5424	0.0669	0.0759	**0.0390**

Table 3.3: Inference speeds of all methods.

Bibliography

[Kat18] Ayoosh Kathuria, *What's new in yolo v3?*, Apr 2018.

[KNF+14] Tomáš Krajník, Matías Nitsche, Jan Faigl, Petr Vanundefinedk, Martin Saska, Libor Přeučil, Tom Duckett, and Marta Mejail, *A practical multirobot localization system*, J. Intell. Robotics Syst. **76** (2014), no. 3–4, 539–562.

[LHK16] Seoungjun Lee, Dongsoo Har, and Dongsuk Kum, *Drone-assisted disaster management: Finding victims via infrared camera and lidar sensor fusion*, 2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE), 2016, pp. 84–89.

[MK18] Fangchang Ma and Sertac Karaman, *Sparse-to-dense: Depth prediction from sparse depth samples and a single image*, 2018.

[ope] *Camera calibration and 3d reconstruction.*

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi, *You only look once: Unified, real-time object detection*, 2016.

[RF18] Joseph Redmon and Ali Farhadi, *Yolov3: An incremental improvement*, 2018.

[TBH+19] Anam Tahir, Jari Böling, Mohammad-Hashem Haghbayan, Hannu T. Toivonen, and Juha Plosila, *Swarms of unmanned aerial vehicles — a survey*, Journal of Industrial Information Integration **16** (2019), 100106.

[Tel04] Alexandru Cristian Telea, *An image inpainting technique based on the fast marching method*, Journal of Graphics Tools **9** (2004), 23 – 34.

[VHS19] Matouš Vrba, Daniel Heřt, and Martin Saska, *Onboard marker-less detection and localization of non-cooperating drones for their safe interception by an autonomous aerial system*, IEEE Robotics and Automation Letters **4** (2019), no. 4, 3402–3409.