Faculty of Science, Technology and Communication

Machine Learning-Based Object Situation Awareness Using a Monocular Camera For the Autonomous Navigation of Multirotor Aerial Robots in Populated Areas

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Youssef Bazzaoui

Supervisor:
Jose-Luis Sanchez-Lopez

December 2019

Acknowledgments

This thesis marks the conclusion of my Master's degree in Computer and Information Sciences. First, I would like to thank my Supervisor Jose-Luis Sanchez-Lopez for giving me the chance to explore such an interesting field of research and providing help and advice whenever I needed it, with a lot of patience.

I also want to thank my friends from the University of Luxembourg for their valuable comments and honest interest during these past few months.

Finally, I would like to express my gratitude to my family for providing me with unfailing support and continuous encouragement throughout the process of this research.

Contents

4

I Introduction 5

II Problem Statement and objectives 8

II.1 Problem statement 8

II.2 Objectives 8

III State of the Art 10

III.1 Machine Learning based object recognition, localization and detection 10

III.2 Machine learning based image segmentation 18

III.3 Machine learning based depth estimation from monocular images . 20

III.4 3D Octree: Octomap 25

IV Proposed algorithm and Results 28

IV.1 Overview 28

IV.2 Object detection and image segmentation of monocular images . . 28

IV.3 Image depth computation 35

IV.4 Point Cloud computation 43

IV.5 Octomap computation 47

V Conclusion and future works 50

I Introduction

Unmanned aerial vehicles (UAV) have increasingly gained in popularity in the recent years (Cuerno-Rejado et al. (2016)). These aircrafts without a human pilot can either be remote controlled or remote guided vehicles, or they can be autonomous vehicles which are capable of sensing their environment and navigating on their own. UAV innovations started in the early 1900s and originally focused on providing practice targets for training military personnel. Throughout the 20th century, they demonstrated the possibility of cheaper, more capable fighting machines, deployable without risk to aircrews. Progressively, in addition to their military applications, UAVs became also used for other aims such as logistics, research and development, agriculture, aerial photography, data collection, topographic mapping, etc. Also in its civilian uses, this technology showed some interesting benefits. For example, in cases of emergency such as natural disasters (earthquakes, volcanoes, nuclear disaster, ...), UAVs have proved useful to assess damage, provide help, locate victims without putting any life in danger. In the environmental field, drones combined with geospatial imagery proved useful in tracking, studying animals, and preventing poaching. And a better supervision of animal life has in turn a positive impact on field epidemiology.

There are several types of UAVs. Fixed-wing UAVs (as opposed to 'rotary wing'such as helicopters) use a wing like a normal aeroplane to provide the lift rather than vertical lift rotors. They are characterized by a fast flight speed, can cover larger areas and have long endurance. The main drawbacks of fixed-wing UAVs are however the cost and the absence of hovering and VTOL. Because they need to be moving in order to be lifted, space is needed to fly and recover the vehicle. Helicopters are not frequently used in unmanned aviation. They are more efficient for hovering with a heavy payload which is why they are typically used for Aerial LIDAR laser scanning. They are also expensive (more than the fixed-wings) and difficult to fly. Conversely, multirotor vehicles are the cheapest, most accessible and easiest to use drones. They have better camera control and can operate in a confined area. Their main disadvantages are the small payload capacity, the limited speed and the shortness of flight time (due to the amount of energy needed and the fact that they use electric motors). They are typically used for aerial

Photography and Video Aerial Inspection. Over the last few years, there have been a huge development concerning the multirotor aerial robots. In the past, cameras hung from cables were used to capture aerial views. Nowadays, multirotor UAVs are being commonly used to capture addresses by politicians, concerts, and live sports events. They are also increasingly used for other tasks such as package delivery thereby allowing for a reduction of risks related to human factors (e.g. the risk of errors in package delivery or delay resulting from traffic congestion).

Although most unmanned aerial robots possess some level of autonomy, the development of fully autonomous technology still faces some challenges. Autonomous systems existing today usually involve some kind of manned intervention whether it is for controlling operation or distance monitoring of the system. To increase autonomy, the system should first be able to collect and properly use information about its environment; otherwise it may cause accidents. That is why situation awareness is a critical factor. Situation awareness does not only include environment awareness and spatial orientation, but also awareness of time, robot configuration and flight control system modes. To facilitate it, appropriate technology must be used, including cameras and computer vision tools.

Most autonomous drones use monocular cameras. A monocular camera is a vision sensor that detects objects and lane boundaries. It is true that stereo cameras can perform better than monocular cameras depth estimation[1]. But the use of the monocular camera reduces the weight and size of robots, and reduces the cost. The processing of images caught by monocular cameras requires the use of Computer Vision and Machine Learning algorithms in order to automatically detect objects and recognize their respective positions in near real-time. More particularly, these models attempt to achieve capabilities that are as close as possible to a human visual system by computationally reconstructing a scene's geometry from 2D image data.

Finally, the Robot Operating System (ROS) is heavily utilized in robotics. The Robot Operating System (ROS) is an open-source flexible framework for writing robot software. It aims to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS contains a set of software libraries

[1] As opposed to monocular cameras, stereo cameras have the advantage of stereoscopic vision: overlapping visual fields by blending slightly dissimilar views of an object (stereo means "solid" or "three-dimensional"). This is the same advantage supplied by binocular vision (with two eyes, an object is perceived through 2 images, allowing for a precise depth perception).

and tools (drivers, state-of-the-art algorithms, etc) and provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between devices and 3D visualization tool. It was built from the ground up to encourage collaborative robotics software development.

II Problem Statement and objectives

II.1 Problem statement

Although the use of drones has become widespread, developing a fully autonomous drone still remains challenging. To make autonomous drones, there are two possibilities. The first one is to include many captors (laser, infrared,..) with real time computer vision calculation. However, this will make drones extremely costly, restricting the usage to big industries (such as military drones). The other possibility is to include only the necessary cheap equipment (in terms of price, weight and battery consumption) such as monocular cameras.This option is more challenging since we have to compensate the absence of advanced equipment with cutting edge computer vision technology. Knowing the environment is crucial to avoid accident (specially static objects like buildings or trees). However, in a minimal solution (small drone with monocular vision, and small cpu and storage), the only sensor available is vision, and it cannot be used intensely since it is battery and time consuming. Therefore, there is a need for a smart equilibrium between frame rates from cameras and the efficiency of algorithms to detect objects, in order to reduce the risk of collusion while keeping time and battery consumption at a minimal level. Most of the existing situational awareness solutions so far were based on stereo cameras. But since machine learning techniques are progressing a lot lately, addressing the problems of image segmentation, object recognition and localization and depth estimation from monocular cameras has also become possible.

II.2 Objectives

The aim of this project is to conceive a solution for situation awareness using monocular cameras. To achieve this goal, we need to fulfill the following objectives:

- Determine the right algorithm for image classification and object detection
  - Explore the state of art for computer vision techniques
  - Consult previous analyses and comparisons of existing image classification and object detection algorithms
  - Test different algorithms and compare them in terms of accuracy and execution

time

  - Determine the appropriate algorithms to use for image segmentation and object detection
  - Combine the chosen algorithms

- Include the depth estimation model
  - Examine the existing algorithms of depth estimation for monocular vision
  - Test different algorithms and compare their performances
  - Choose the right algorithm
  - Verify the obtained disparity images and make the necessary adjustments to the algorithm

- Create a 3d map based on the obtained disparity images
  - Implement an algorithm for point cloud computation based on disparity images
  - Compare the existing solutions for building a 3D map
  - Convert the point clouds to a 3D map

III State of the Art

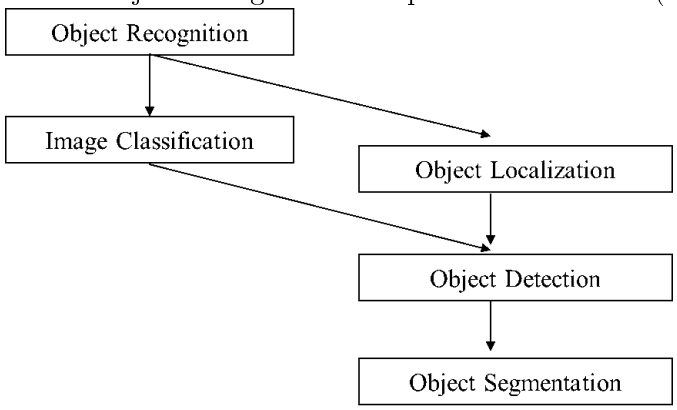
III.1 Machine Learning based object recognition, localization and detection

III.1.1 General definitions

Object recognition is a general term to describe a collection of related computer vision tasks that involve identifying objects in digital images or frames. It involves the following three computer vision tasks (Brownlee (2019)):

- Image classification : a technique that predicts the type or class of an object in an image
- Object localization: locates the presence of objects in an image and indicates their location with a bounding box
- Object detection: a technique that locates objects with a bounding box and types or classes of the identified objects in the image.

Figure 1: Overview of Object Recognition Computer Vision Tasks (Brownlee (2019))



As shown on figure 1, an additional computer vision task is object segmentation (or semantic segmentation) which consists in highlighting pixels of instances from objects instead of using a bounding box (see example on figure 8 from the following section).

Based on work related to neural networks and learning systems, neural network algorithms have developed in the recent years with a significant impact on image classification and object detection techniques as learning systems. CNNs are multi-layer

neural networks designed to recognize visual patterns directly from pixel images with minimal preprocessing. They are different from traditional approaches as they have the ability to learn more complex features and the training algorithms allow to learn object representations without the need to design features manually (Zhao et al. (2019)).

III.1.2 Image classification

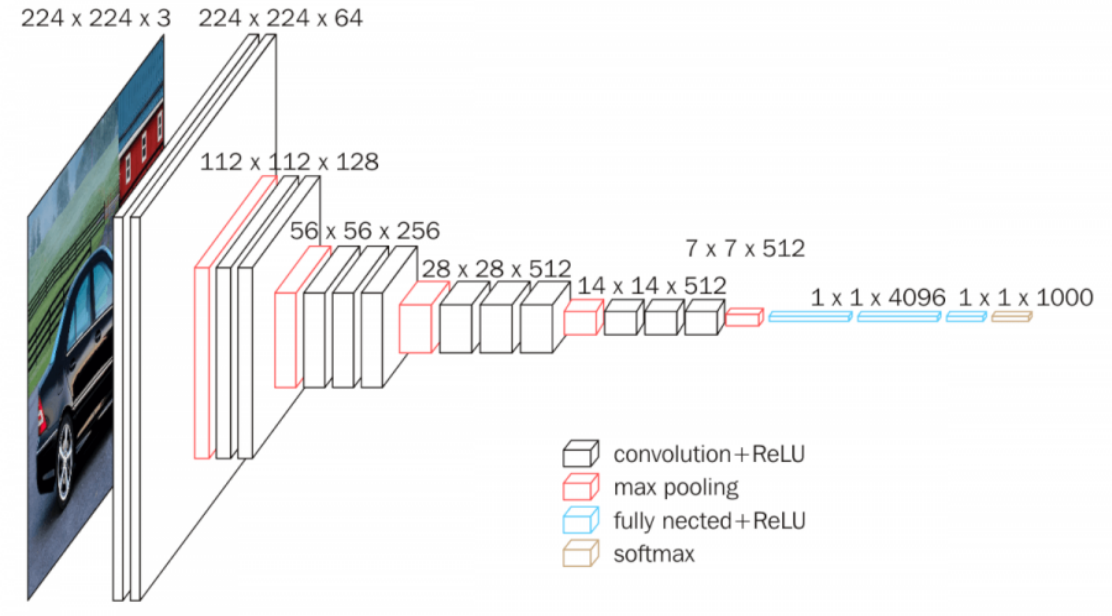
Image classification is a technique that predicts the type or class of an object in an image. AlexNet was the pioneering deep CNN for image classification that won the ILSVRC-2012 with a Top-5 test accuracy of 84.6%. The architecture consists of five convolutional layers, max-pooling ones, Rectified Linear units (RELUs) as non-linearities, three fully-connected layers and dropouts Krizhevsky et al. (2012). Another typical CNN architecture is called Visual Geometry Group (VGG), a CNN model introduced by the Visual Geometry Group (VGG) from the University of Oxford. One of these models, VGG16 is represented in the following figure (proposed by Simonyan and Zisserman (2014)). [2]

The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. Different types of transformations are conducted such as filtering (convolution) and pooling (such as max pooling and L2 pooling). The typical VGG16 has totally 13 convolutional layers, 3 fully connected layers, 3 max-pooling layers and a softmax classification layer.

GoogLeNet is a rather complex CNN architecture introduced by Szegedy et al. (2015), composed of 22 layers, and a newly introduced building block called an inception module. This module reduces the number of parameters and operations thereby allowing for a significant cost and memory saving. Microsoft ResNet consists of 152 layers in addition to residual blocks. A residual neural network (ResNet) is an artificial neural network (ANN) of a kind that builds on constructs known from pyramidal cells in the cerebral

[2] The input to cov1 layer is of fixed size $224 * 224$ RGB image. The image is passed through a stack of convolutional layers, where the filters are used with a very small receptive field: $3 * 3$. The convolution stride is fixed to 1 pixel; the spatial padding of convolutional layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for $3 * 3$ convolutional layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the convolutional layers. Max-pooling is performed over a $2 * 2$ pixel window, with stride 2. Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks. See link `https://neurohive.io/en/popular-networks/vgg16/`

Figure 2: VGG16



cortex. Residual neural networks do this by utilizing skip connections, or shortcuts to jump over some layers.

MDRNN was suggested by Graves et al. (2007) to extend recurrent neural networks (RNNs) architectures to multi-dimensional tasks. Based on this approach,Visin et al. (2015) proposed ReNet, an architecture that replaces the ubiquitous convolution and pooling layer of the deep convolutional neural network with four recurrent neural networks that sweep horizontally and vertically in both directions across the image.

III.1.3 Object localization

Object localization is a technique that consists in locating the presence of objects in an image and indicating their location with a bounding box. Even though object detection and object localization are close concepts, they are very different. Object localization algorithms label the class of a single object, and draw a bounding box around the position of that object. Object detection on the other hand is needed when there are multiple objects on an image. In other words, localization basically focus in locating the most visible object in an image while object detection focus in searching out all the objects and their boundaries. In object localization, one convolutional neural network can be used and trained to generate the coordinates of the object in question. The same convolution

network as that for image classification can be used, by changing the neural network to have a few more output units that contain a bounding box. Based on regression techniques, the bounding box can be drawn using 4 numbers (x_0,y_0, width, height) (Grover (2018)).

III.1.4 Class and instance object detection

As stated before, object detection refers to the task of estimating the concepts and locations of objects contained in each image. It consists in determining both where objects are located in a given image and to which category they belong. Traditional object detection models are divided into three stages:

- Informative region selection: to specify the exact position of an object, scanning the whole image with a multi-scale window would permit to find out all possible positions of the objects. However, this approach is computationally expensive, especially that many redundant windows could be generated. At the same time, if the number of generated windows is set, the result could be unsatisfactory.
- Feature extraction: due to the diversity of appearances, luminosity conditions, and backgrounds, designing a descriptor that would be able to extract visual features and recognize all objects is not easy
- Classification: a classifier is needed to distinguish an object based on hierarchical representations.

The main application domains of object detection are as follows:

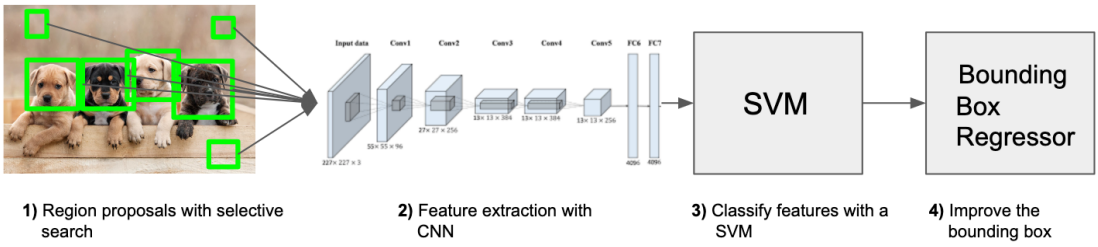
- Generic object detection : achieved with bounding box regression
- Salient object detection : achieved with local contrast enhancement and pixel-level segmentation
- Face detection and pedestrian detection: mainly accomplished with multi-scale adaption and multi-feature fusion/boosting forest

Generic object detection aims at locating and classifying existing objects in any one image, and labeling them with rectangular bounding boxes to show the confidence intervals of existence. We distinguish between two types of generic object detection models:

one that follows a traditional approach, first generating region proposals then classifying these proposals into object categories. These Region-based methods include R-CNN, Fast R-CNN, Faster R-CNN, and Mask R-CNN, among others. The second type regards object detection as a regression or classification problem, adopting a unified framework; that is the case of YOLO for example.

- R-CNN: this object detection system was suggested by Ross Girshick in 2014 and obtained a mean average precision of 53.3% with more than 30% improvement over the previous best results. The idea was to take the input image and divide it into 2000 regions called region proposals using a selective algorithm. The searching method provides more accurate candidate boxes while reducing the searching space. Then features extraction is made using a convolutional neural network (CNN). A 4096-dimensional feature is extracted as the final representation. Then each region is classified using class-specific linear SVMs. The scored regions are finally adjusted with bounding box regression and filtered with a greedy non-maximum suppression (NMS) to produce final bounding boxes for preserved object locations. Since the

Figure 3: The R-CNN system

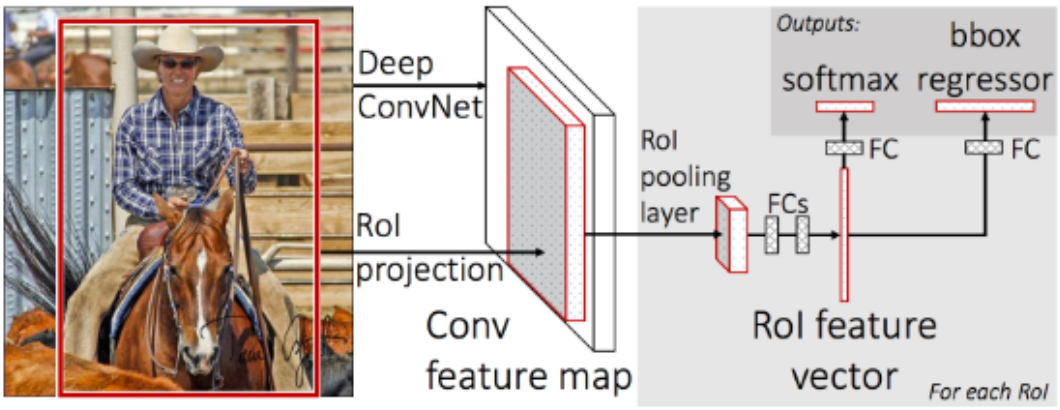


development of R-CNN, many improved models have been suggested such as Fast R-CNN which jointly optimizes classification and bound-box regression tasks, Faster R-CNN which takes an additional subnetwork to generate region proposals and YOLO which accomplishes object detection via fixed-grid regression.

- Fast R-CNN: In a subsequent paper, R. Girshick proposed a new training algorithm that fixes or counters a few drawbacks of R-CNN by improving the speed and the accuracy. Fast R-CNN takes as input an entire image and a set of object proposals, and then processes the whole image with the CNN. To produce a convolutional feature map and then for each region proposal they extract the corresponding

point from the feature map which is called region proposed feature map. Then for each object proposal a region of interest (ROI) pooling layer is extracted with a fixed length feature vector from feature maps. Then each vector is fully fed into a sequence of fully connected layers that have two output layers. The first produces the SoftMax probability for all object categories (in addition to a background class). The other output layer encodes refined bounding-box positions with four real-valued numbers.

Figure 4: Fast R-CNN

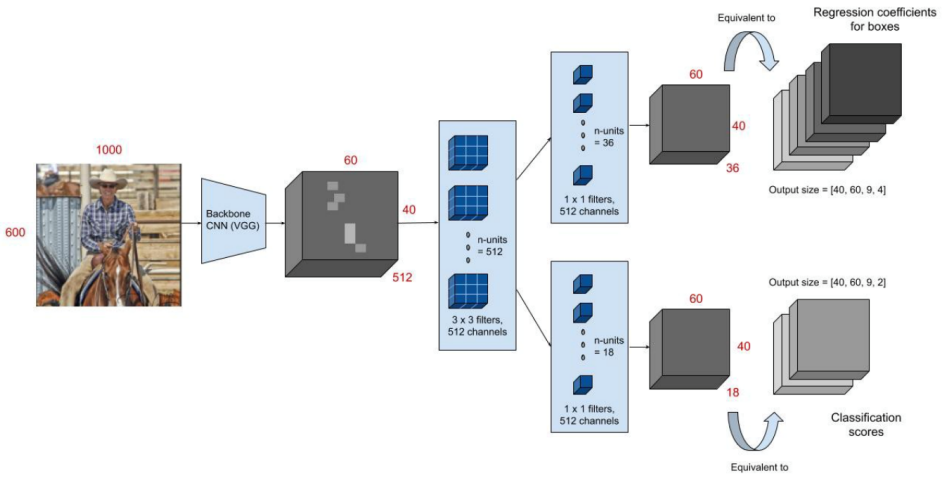


- Faster R-CNN: Uses a convolutional network instead of using a selective search algorithm which is a slow process and takes around 2 second per image. It consists of two stages: the first stage, called a Region Proposal Network (RPN), proposes candidate object bounding boxes. The second stage (which is basically Fast R-CNN) extracts features using RolPool from each candidate box and performs classification and bounding-box regression.

- Mask R-CNN: it adopts the same two-stage approach as Faster R-CNN with an identical first stage. In the second stage however, in parallel to predicting the class and box offset, Mask R-CNN also outputs a binary mask for each RoI. The following items depict the general steps the approach follows:

  – Backbone model: a standard convolutional neural network that serves as a feature extractor. For example, it will turn a1024x1024x3 image into a 32x32x2048 feature map that serves as input for the next layers.

Figure 5: Faster R-CNN

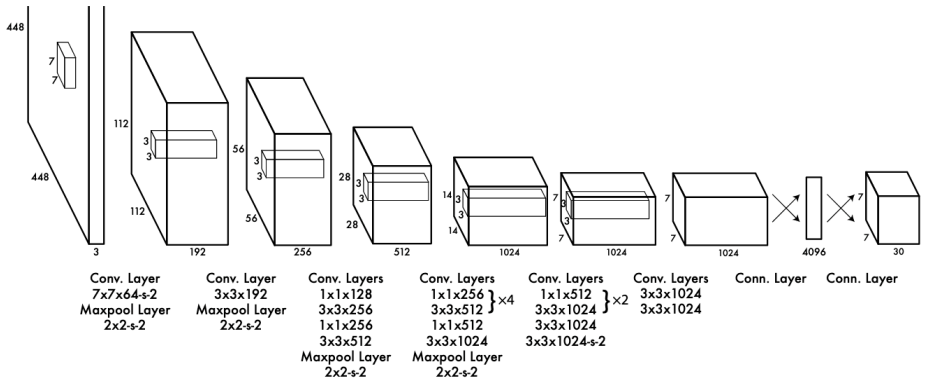


  - Region Proposal Network (RPN): Using regions defined with as many as 200K anchor boxes, the RPN scans each region and predicts whether or not an object is present.
  - Region of Interest Classification and Bounding Box: In this step the algorithm takes the regions of interest proposed by the RPN as inputs and outputs a classification (softmax) and a bounding box (regressor).
  - Segmentation Masks: In the final step, the positive ROI regions are taken in as inputs and 28x28 pixel masks with float values are generated as outputs for the objects.

- You Only Look Once (YOLO): Yolo is an algorithm based on regression, instead of selecting only interesting regions of an image, Yolo predict classes and bounding boxes for the whole image. Therefore, using this system, you only look once at an image to predict what objects are present and where they are. It applies a single neural network to the full image. The network uses features from the entire image to predict each bounding box. It consists of 24 convolutional layers followed by 2 fully connected layers.

The input image is divided into S*S, if the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the

Figure 6: Network architecture (YOLO)

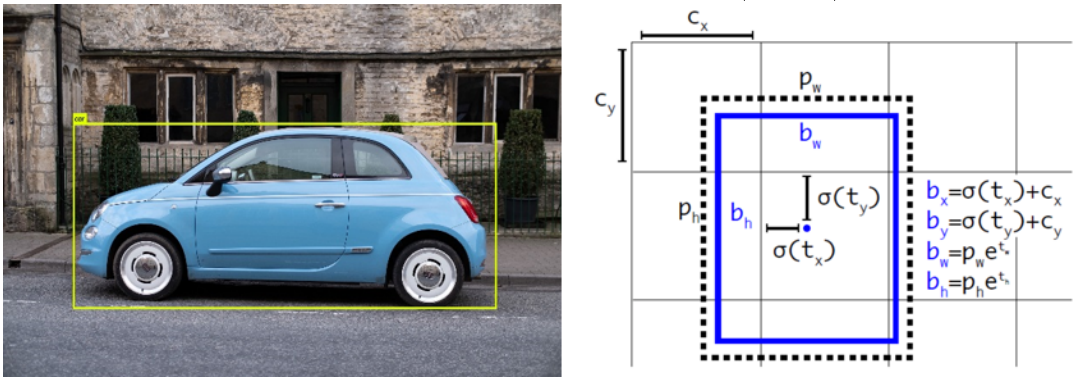


model is that the box contains an object. The confidence is defined as P(object). For each bounding box, the network predicts 4 coordinates t_x, t_y, t_w, t_h. If the cell is offset from the top left corner of the image $b_y\,(c_x, c_y)$ and the bounding box prior has width and height p_w, p_h, then the predictions correspond to:

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^t$$
$$b_h = p_h e^t$$

One of the advantages of YOLO is that it is fast compared to other algorithms, but it has some limitations when it comes to detecting small objects. YOLO achieves a significantly

Figure 7: Bounding-box predictions (YOLO)



high performance in comparison to other detection methods. More particularly, YOLOv3 runs significantly faster than other methods with comparable performance as shown on

the following table.

Table 1: Comparison between YOLOv3 and other detectors, in terms of inference time and mAP Redmon and Farhadi (2018)

Method	mAP	Time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	37.8	198
YOLOv3-320	28.2	22
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

III.2 Machine learning based image segmentation

III.2.1 A general definition of image segmentation

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels). The goal of segmentation is to simplify or change the representation of an image into something that is more meaningful and easier to analyze.

III.2.2 Class Vs Instance segmentation

Semantic segmentation is a computer vision problem that performs per-pixel labeling of objects at class level, where segmented objects are usually represented by masks. Performing this task can be challenging due to cluttered backgrounds, deformed objects, different scales, etc. One pioneering approach is the Fully Convolutional Network (FCN) by Long et al. (2015). Fully Convolutional Networks (FCNs) owe their name to their architecture, which is built only from locally connected layers, such as convolution, pooling and upsampling. Note that no dense layer is used in this kind of architecture. The FCN is a normal CNN, where the last fully connected layer is substituted by another convolution layer with a large "receptive field". The idea is to capture the global context of the scene. This reduces the number of parameters and computation time. To obtain a segmentation map (output), segmentation networks usually have 2 parts:

Table 2: Deep learning semantic segmentation methods

Decoder Variant	SegNet, Bayesian SegNet	
Integrating Context Knowledge	Conditional Random Fields	DeepLab, CRFas RNN, MINC-CNN
	Dilated Convolutions	Dilation, ENet
	Multi-scale Prediction	Multi-scale CNN-Raj, Multi-scale CNN-Bian, Multi-scale CNN-Eigen, Multi-scale CNN-Roy
	Recurrent Neural Networks	DAG-RNN, rCNN, ReSeg, LSTM-CF, 2D-LSTM
	Feature Fusion	Parse Net, Sharp Mask
Instance segmentation	Deep Mask, SDS, Multi PathNet	
Video Sequences	Clockwork, 3DCNN-Zhang, End2End, Vox2Vox	
3D Data	PointNet, Huang-3DCNN	

- Downsampling path: capture semantic/contextual information
- Upsampling path: recover spatial information

The FCN model still suffered from some shortcomings such as absence of instance-awareness, efficiency issues and inadequacy with unstructured data such as 3D point clouds or models. The following table summarizes some of the main deep learning semantic segmentation methods as discussed by Garcia-Garcia et al. (2017).

While the aim of semantic segmentation is to classify objects features in the image comprised of sets of pixels into meaningful classes that correspond with real-world categories, instance segmentation identifies each instance of each object featured in the image instead of categorizing each pixel. For example, in the image below, instead of classifying all cubes as one instance, it will identify each individual cube. Therefore, the main purpose of instance segmentation is to represent objects of the same class splitted into different instances. Earlier methods of instance segmentation resorted to bottom up segments such as Borenstein and Ullman (2002),Girshick et al. (2014), Hariharan et al. (2015),Dai et al. (2015). The approach suggested by Hariharan et al. (2015) is a Simultaneous Detection and Segmentation (SDS) method which uses a bottom-up hierarchical

Figure 8: Difference between object recognition tasks Garcia-Garcia et al. (2017)

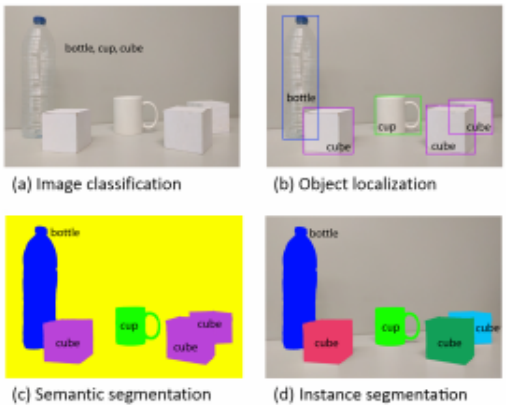


image segmentation along with an object candidate generation process called Multi-scale Combinatorial Grouping (MCG) to obtain region proposals. For each region, features are extracted through an adapted version of R-CNN and each region proposal is classified using a linear Support Vector Machine (SVM) on top of the CNN features. DeepMask Pinheiro et al. (2015) and following works such as SharpMask Pinheiro et al. (2016) learn to propose segment candidates, which are then classified by Fast R-CNN. In these methods, segmentation precedes recognition, which is slow and less accurate. Zagoruyko et al. (2016) proposed the MultiPath classifier system with achieved 66% improvement over the baseline Fast R-CNN.

III.3 Machine learning based depth estimation from monocular images

Most existing depth estimation approaches rely on supervised machine learning methods. These methods require large quantities of ground truth depth data for training. And the vast majority of this body of work has used strong assumptions about the scene geometry of the environment or multiple image geometric cues to infer depth (e.g. stereopsis). However depth estimation in case of a single-image is more challenging and a fundamental problem in computer vision. First models in this regard do not apply to surfaces with a uniform color and texture, they include for example shape-from-shading and shape-from-texture. Nagai et al. (2002) used Hidden Markov Models to perform surface reconstruction from single images from known, fixed objects such as hands and faces. Han and Zhu (2003) performed 3-d reconstruction for known specific classes of objects placed in untextured areas. Saxena et al. (2006, 2008) suggested an algorithm

for predicting depth from monocular image features, which appeared to be particularly useful in the case of stereovision. In a more recent work, Saxena et al. (2009) presented a patch-based model known as Make3D. This approach consists in over-segmenting the image into many small homogeneous regions called "superpixels" and then using a Markov Random Field (MRF) to infer the 3-d position and orientation of each. No assumptions were made regarding the structure of the scene which allowed for a good generalization of the model, even to scenes with significant non-vertical structure.

This method was also used to build 3-d models based on a sparse set of images and using object recognition information. It proved to be more efficient than previous models, both in terms of quantitative accuracies in predicting depth and in terms of fraction of qualitatively correct models. However, as pointed out by Godard et al. (2017), this method along with most planar based approximations face many difficulties when modeling thin structures, and lack the global context needed to generate a realistic output.

Ladicky et al. (2014) proposed a new pixel-wise classifier that can jointly predict a semantic class and a depth label from a single image. By exploiting the property of perspective geometry, authors reduced the learning of a pixel-wise depth classifier to a much simpler classifier predicting only canonical depth, thereby removing the training dataset biases. The approach also led to quantitatively better results in the semantic segmentation domain through the use of proper alignment. However, as pointed out by the authors, the method suffers from several weaknesses, namely the inability to deal with low resolution images, very large requirements in terms of hardware and apparent inability to locate the objects more precisely for semantic classes with high variance.

Karsch et al. (2014) suggested a fully automatic technique to estimate depths for videos using non-parametric depth sampling and also applied it to single images. For videos, local motion cues were used to improve the inferred depth maps, while optical flow was used to ensure temporal depth consistency. For training and evaluation, a large dataset containing stereoscopic videos with known depths was collected. This technique proved effective in cases where past methods failed (non-translating cameras and dynamic scenes) and made it possible to generate stereoscopic videos for 3D viewing from conventional 2D videos. The single image algorithm also outperformed previous approaches.

However, the main constraint is that whole depth images from a training dataset had to be copied before and be available at test time in order to achieve the consistency of image level predictions.

Liu et al. (2015) criticized the insufficient efficiency and flexibility of depth prediction in new images prevailing in traditional MRF methods. As underlined by the authors, these approaches require more time and rely on the horizontal alignment of images and the semantic labellings of the training data available before-hand. They instead suggested a continuous convolutional neural network (CNN) approach. Initially, convolutional networks were applied in object classification and detection in works such as (Girshick et al. (2014), Krizhevsky et al. (2012), Sermanet et al. (2013),Szegedy et al. (2015),Jaderberg et al. (2015)). Then other systems developed using those networks for other tasks such as pose estimation, stereo depth, and instance segmentation. The model suggested by Liu et al. (2015) was a deep continuous CRF model in which the log-likelihood optimization problem could be solved directly with no approximations and where CNN was used to construct unary and pairwise potentials of the CRF, as opposed to the hand-crafted features adopted in previous work.

Eigen et al. (2014) proposed a similar system based on a multi-scale convolutional network with one major difference is that they regressed directly the depth map from an input image through convolutions, whereas in contrast Liu et al. (2015) used a CRF to explicitly model the relations of neighboring super-pixels and learn the potentials in a unified CNN framework.

In a subsequent work, Eigen et al. (2014) built upon the previous model to develop a more general network that uses a sequence of three scales to generate features and refine predictions to higher resolution. This approach was applied to multiple tasks, including surface normal estimation and per-pixel semantic labeling. This method does not rely on any superpixels or low-level segmentation.

Several works have been built based on this approach using techniques such as CRFs to improve accuracy. For example, Wang et al. (2015) combined the CNNs with CRFs, they formulated depth estimation in a two-layer hierarchical CRF to enforce synergy between global and local predictions. Roy and Todorovic (2016) proposed a neural regression forest (NRF) architecture which combines convolutional neural networks with

random forests for predicting depths in the continuous domain via regression. Cao et al. (2017) presented an approach which formulates depth estimation as a pixel-wise classification task instead of regression. This allowed them to obtain the confidence of a depth prediction in the form of probability distribution. With this confidence, they could apply an information gain loss to make use of the predictions that are close to ground-truth during training, as well as fully-connected conditional random fields (CRF) for post-processing to further improve the performance.

Nevertheless, as the main limit of supervised methods is that they require substantial ground truth data, other novel methods have been suggested based on an unsupervised approach. Much of these recent methods used a combination of 3D shape with image warping and blending, such as Eisemann et al. (2008), Goesele et al. (2010), Chaurasia et al. (2011), Chaurasia et al. (2013).

Flynn et al. (2016) introduced a novel image synthesis network called DeepStereo that does not require known depth or disparity as training data. This approach was based on the work of Fitzgibbon et al. (2005) on IBR using image-based priors. Authors stressed that the underlying idea of this work is that the goal of faithfully reconstructing the output image should be the key problem to be optimized for, as opposed to reconstructing depth or other intermediate representation. The model of Flynn et al. (2016) generates new views by selecting pixels from nearby images. The pixels from neighboring views of a scene are presented to the network, which then directly produces the pixels of the unseen view. The benefits of this approach include generality and high quality results on traditionally difficult scenes mainly due to the fact that it is trained end-to-end. Pixels are generated according to color, depth, and texture priors learnt automatically from the training data. Image synthesis is performed on small overlapping patches. One drawback of this method is speed, which is minutes per image. Also, it requires reprojecting each input image to a set of depth planes which limits the resolution of the output images that can be produced. In addition, as several nearby posed images are necessary at test time, DeepStereo is not suitable for monocular depth estimation.

Building on the work of Flynn et al. (2016), Xie et al. (2016) used the same probabilistic selection layer of DeepStero to propose a fully automatic 2D-to-3D conversion algorithm based on deep convolutional neural networks. Their method was trained end-

to-end on stereo image pairs directly, thus able to exploit more orders of magnitude of data than traditional learning based 2D-to-3D conversion methods. Their goal was to generate the corresponding right view from an input left image. The resulting synthesized right image pixel values are a combination of the pixels on the same scan line from the left image, weighted by the probability of each disparity. They improve upon the approach of Flynn et al. (2016) in two ways. First, they remove the limitation related to the need for more than one calibrated view to synthesize a novel view, by restructuring the network input and layout. Second, they develop an approach that does not rely on small patches but processes the entire image, allowing large receptive fields necessary to take advantage of high-level abstractions and regularities. However, the main drawback of their model is that increasing the number of candidate disparity values greatly increases the memory consumption of the algorithm, making it difficult to scale their approach to bigger output resolutions.

To overcome the limit of ground-truth data, Garg et al. (2016) presented the first convolutional neural network for single-view depth estimation that can be trained end-to-end from scratch, in a fully unsupervised fashion, simply using data captured using a stereo rig. The suggested approach was based on training the network in a manner analogous to an auto-encoder. The source and target of this method are a pair of images, with small, known camera motion between the two such as a stereo pair. The convolutional encoder was trained for the task of predicting the depth map for the source image, by explicitly generating an inverse warp of the target image using the predicted depth and known inter-view displacement, in order to reconstruct the source image. The photometric error in the reconstruction was equivalent to the reconstruction loss for the encoder. However, their image formation model was not fully differentiable, as underlined by Godard et al. (2017). To compensate, they performed a Taylor approximation to linearize their loss resulting in an objective that is more challenging to optimize. To overcome this problem, Godard et al. (2017) suggested a fully convolutional deep neural network model for image reconstruction that uses bilinear sampling to generate images, resulting in a fully (sub-) differentiable training loss. Exploiting epipolar geometry constraints, authors suggested a novel training image reconstruction loss that enforces consistency between the disparities produced relative to both the left and right images, leading to

improved performance and robustness compared to existing approaches.

III.4 3D Octree: Octomap

In order to encode the information of the environments that the robot can use to distinguish different terrains, localize itself, and plan a safe trajectory, maps have to be built, usually using the ultrasound sensor and the laser range finder. Oftentimes, these sensors are mounted on robot platforms to build a 2D map of the world. However, it is not sufficient to represent the 3D world just relying on one slice of the environment, especially when the robot is operating in an uneven and unstructured environment. Recently, sensors with 3D environment sensing capabilities have become increasingly common and much efforts have been dedicated to building a precise 3D map with these sensors.

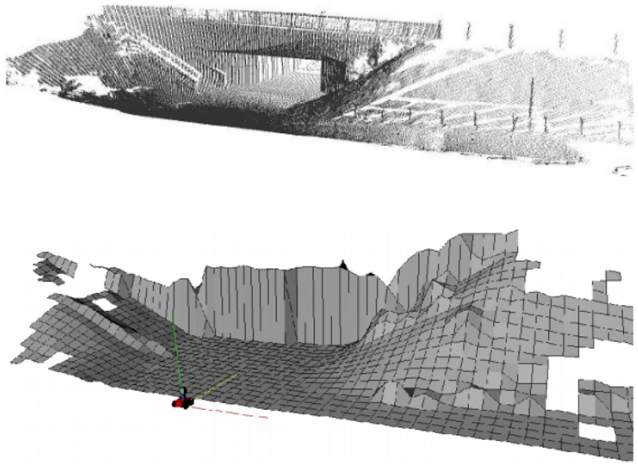
One popular approach is the use of a grid of cubic volumes of equal size, called voxels, to discretize the mapped area. The main limit of this method is the excessive memory requirement, especially in the case of large-scale outdoor scenarios or when there is the need for fine resolutions. In addition, the extent of the map area needs to be known beforehand; otherwise costly copy operations will be performed each time the extent of the map is changed.

The drawbacks of this method can be avoided by storing 3D range measurements directly and then modeling the occupied space by the 3D point clouds returned by range sensors (laser finders, stereo cameras, etc). For example, in Cole and Newman (2006) a novel 3D SLAM algorithm adopting a 3D laser scanner is proposed. The world is represented by point clouds and the real-time interpretation and update of 3D data still pose great challenges for low power laptops, and these slow updates result in the robot either moving slowly or being in a potentially unsafe manner. In addition, the free and occupied spaces are not modeled clearly. Therefore, point clouds can only be suitable for high precision sensors in static environments and when unknown areas do not need to be represented.

According to Hornung et al. (2013), conditional on some assumptions about the mapped area, 2.5D maps are sufficient to model the environment. The main advantages of this method are memory efficiency and constant time access, the main shortcomings is that it is a non-probabilistic approach and there is no distinction between free and

unknown space. When there is a single surface that the robot uses for navigation, the elevation maps are an appropriate representation because overhanging objects (trees, bridges, etc) can be ignored. A 2D grid stores an estimated height (elevation) for each cell, instead of the occupancy within the area that the grid patch represents. Terrain characteristics like height difference, slope and roughness can be efficiently assessed from grids' average height instead of directly from dense sensor feedback from stereovision, which will speed the functions. One fundamental limit of elevation maps is that only one level is represented and vertical objects are excluded. Ryde and Hu (2010) suggested a volumetric approach which stores a list of occupied voxels for each cell in a 2D grid. However, this representation does not differentiate between free and unknown volumes.

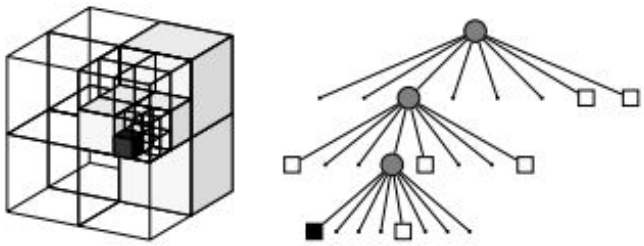
Figure 9: A typical elevation map for a bridgePfaff et al. (2007)



To cope with these shortcomings, a more compact 3D environment representation, the Octomap, was suggested for autonomous legged robot navigation. As opposed to methods based on a fixed grid structure, the extent of the mapped environment does not need to be known beforehand and the map only contains volumes that have been measured. Octomap's approach is based on octrees and uses probabilistic occupancy estimation. An octree is a hierarchical data structure for spatial subdivision in 3D. Each node in an octree represents the space contained in a cubic volume (voxel).This volume is recursively subdivided into eight sub-volumes until a given minimum voxel size is reached, as shown on the following figure.

Octrees can be used to model a Boolean property. If a certain volume is measured

Figure 10: Example of an octree storing free shaded white and occupied black cells (from left to right: the volumetric model and the tree representation), Hornung et al. (2013)



as occupied, the corresponding node in the octree is initialized. Any unitinalized node should be either free or unknown. If free volumes are represented in the tree between the sensor and the measured end point, areas that are not initialized model unknown space. Using Boolean occupancy states allows for compact representation of the octree. However, in robotic systems, a discrete occupancy label is not sufficient due to sensor noise and the environment changes. Therefore, occupancy has to be modeled probabilistically. In this approach, the occupancy grid mapping method is used such that the probability of a leaf node to be occupied given the sensor measurements is estimated based on

$$P\left(n|z_{1:t}\right) = \left[1 + \frac{1-P(n|z_t)}{P(n|z_t)} \frac{1-P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1-P(n)}\right]^{-1}$$

The common assumption of a uniform prior probability leads to P(n)= 0.5 and by using the log-odds notation, this equation can be rewritten as:

$$L\left(n|z_{1:t}\right) = L\left(n|z_{1:t-1}\right) + L\left(n|z_t\right)$$

such that

$$L\left(n\right) = log\left[\frac{P(n)}{1-P(n)}\right]$$

This formulation of the update rule allows for faster updates since multiplications are replaced by additions.

Octomap is available as an open-source C++ library, and has already been successfully applied in several robotic projects. As an application example of this framework, Hornung et al. (2012) divide the Octomap into multiple layers, in which the planning of the robot mobile base and the manipulator are performed on different map layers to improve planning efficiency.

IV Proposed algorithm and Results

IV.1 Overview

To conduct this project, we went through the following steps

- First, we compared different algorithms of object detection and image segmentation
- Then, we chose the more adequate algorithms based on considerations of speed and accuracy
- In a subsequent step, we identified the algorithm that should be used for depth estimation and adapted this algorithm based on the images we used for testing
- We converted the obtained disparity images to point clouds
- In a final step, we used Octomap to convert the point clouds to octrees and were able to visualize the result in Octomap Viewer

IV.2 Object detection and image segmentation of monocular images

IV.2.1 Description

In terms of object detection, we tried two approaches: YOLO V3 and Mask R-CNN. The training dataset was Coco, as an initial attempt, even though it is not sufficient for drone manipulation (since the list of objects in the Coco dataset is limited: `http://cocodataset.org/#explore`). The main requirements to use YOLO are Python 3.7 or a later version and Numpy. The main motivation behind our first choice are the previous studies of comparison between existing algorithms in terms of speed and accuracy[3].

Since the aim of our project is to design an algorithm fit for drones, speed of object detection was an important stake, and YOLO provided a satisfactory result in this respect. However, the main shortcoming of this approach is the lack of segmentation. It could not detect small objects and its level of accuracy was not sufficient to create a 3D map (a 3d object model was represented as a cuboid). In addition, when YOLO was combined with a semantic segmentation algorithm, its efficiency was undermined (slower

[3] See link `https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison\-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359`

response). So we decided to use Mask R-CNN, not only for object detection but for segmentation as well, with a script adjustment to include the Monodepth model and to add distance calculation. In order to use Mask R-CNN, we had to install different libraries: numpy, scipy, Pillow, cython, matplotlib, scikit-image, tensorflow>=1.3.0, keras>=2.0.8, opencv-python, h5py, imgaug, IPython[all]. The Mask R-CNN model contains four files:

- frozen_inference_graph.pb : The Mask R-CNN model weights. The weights are pre-trained on the COCO dataset.
- mask_rcnn_inception_v2_coco_2018_01_28.pbtxt: The Mask R-CNN model configuration.
- object_detection_classes_coco.txt: All 90 classes are listed in this text file, one per line.
- colors.txt: This text file contains six colors to randomly assign to objects found in the image.

The Mask R-CNN script performs instance segmentation by applying a mask to the image so that one can see where, down to the pixel, the Mask R-CNN thinks an object is. It contains the following code.

```
# import the necessary packages
import numpy as np
import argparse
import random
import time
import cv2
import os
```

First required packages are imported. Notably, NumPy and OpenCV. Other packages come with most Python installations. Then the command line arguments are parsed.

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-m", "--mask-rcnn", required=True,
    help="base path to mask-rcnn directory")
ap.add_argument("-v", "--visualize", type=int, default=0,
    help="whether or not we are going to visualize each instance")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
ap.add_argument("-t", "--threshold", type=float, default=0.3,
    help="minimum threshold for pixel-wise mask segmentation")
args = vars(ap.parse_args())
```

The following two arguments are required:

–image: The path to our directory input image.

–mask-rnn : The path to the Mask files.

It is also possible to add the probability value through the argument 'confidence' in order to filter weak detection. Another optional argument is threshold (to filter weak mask detection). After this step, we load our labels and colors:

```
# load the COCO class labels our Mask R-CNN was trained on
labelsPath = os.path.sep.join([args["mask_rcnn"],
    "object_detection_classes_coco.txt"])
LABELS = open(labelsPath).read().strip().split("\n")

# load the set of colors that will be used when visualizing a given
# instance segmentation
colorsPath = os.path.sep.join([args["mask_rcnn"], "colors.txt"])
COLORS = open(colorsPath).read().strip().split("\n")
COLORS = [np.array(c.split(",")).astype("int") for c in COLORS]
COLORS = np.array(COLORS, dtype="uint8")
```

The COCO object class LABELS is loaded. Today's Mask R-CNN is capable of recognizing 90 classes including people, vehicles, signs, animals, everyday items, sports gear, kitchen items, food, and more. All available classes can be browsed in the object_detection_classes_coco.txt file. From there COLORS is loaded from the path, performing a couple array conversion operations.

```
# derive the paths to the Mask R-CNN weights and model configuration
weightsPath = os.path.sep.join([args["mask_rcnn"],
    "frozen_inference_graph.pb"])
configPath = os.path.sep.join([args["mask_rcnn"],
    "mask_rcnn_inception_v2_coco_2018_01_28.pbtxt"])

# load our Mask R-CNN trained on the COCO dataset (90 classes)
# from disk
print("[INFO] loading Mask R-CNN from disk...")
net = cv2.dnn.readNetFromTensorflow(weightsPath, configPath)
```

These lines build our weight and configuration paths, followed by loading the model via these paths. In the next block, an image is loaded and passed through the Mask R-CNN neural net:

```
# load our input image and grab its spatial dimensions
image = cv2.imread(args["image"])
(H, W) = image.shape[:2]

# construct a blob from the input image and then perform a forward
# pass of the Mask R-CNN, giving us (1) the bounding box  coordinates
# of the objects in the image along with (2) the pixel-wise segmentation
# for each specific object
blob = cv2.dnn.blobFromImage(image, swapRB=True, crop=False)
net.setInput(blob)
start = time.time()
(boxes, masks) = net.forward(["detection_out_final", "detection_masks"])
end = time.time()

# show timing information and volume information on Mask R-CNN
print("[INFO] Mask R-CNN took {:.6f} seconds".format(end - start))
print("[INFO] boxes shape: {}".format(boxes.shape))
print("[INFO] masks shape: {}".format(masks.shape))
```

First the input image is loaded and dimensions are extracted for scaling purposes later. A blob is constructed via cv2.dnn.blobFromImage. A forward pass of the blob through the net while collecting timestamps is performed. The results are contained in two important variables: boxes and masks. After a forward pass of the Mask R-CNN is performed on the image, the following code is used to filter + visualize the results:

```
# loop over the number of detected objects
for i in range(0, boxes.shape[2]):
    # extract the class ID of the detection along with the confidence
    # (i.e., probability) associated with the prediction
    classID = int(boxes[0, 0, i, 1])
    confidence = boxes[0, 0, i, 2]

    # filter out weak predictions by ensuring the detected probability
    # is greater than the minimum probability
    if confidence > args["confidence"]:
        # clone our original image so we can draw on it
        clone = image.copy()

        # scale the bounding box coordinates back relative to the
        # size of the image and then compute the width and the height
        # of the bounding box
        box = boxes[0, 0, i, 3:7] * np.array([W, H, W, H])
        (startX, startY, endX, endY) = box.astype("int")
        boxW = endX - startX
        boxH = endY - startY
```

In this block, the filter/visualization loop starts. We proceed to extract the classID and confidence of a particular detected object. From there we filter out weak predictions by comparing the confidence to the command line argument confidence value, ensuring we exceed it. Assuming that is the case, we will go ahead and make a clone of the image since we will need this image later. Then we scale our object's bounding box as well as calculate the box dimensions.

```
# extract the pixel-wise segmentation for the object, resize
# the mask such that it's the same dimensions of the bounding
# box, and then finally threshold to create a *binary* mask
mask = masks[i, classID]
mask = cv2.resize(mask, (boxW, boxH),
    interpolation=cv2.INTER_NEAREST)
mask = (mask > args["threshold"])

# extract the ROI of the image
roi = clone[startY:endY, startX:endX]
```

The pixel-wise segmentation is extracted for the object which is resized to the original image dimensions. Then a threshold is applied to the mask so that it is a binary array/image. We also extract the region of interest where the object resides. For convenience, this next block accomplishes visualizing the mask roi and segmented instance if the –visualize flag is set via command line arguments:

```
# check to see if are going to visualize how to extract the
# masked region itself
if args["visualize"] > 0:
    # convert the mask from a boolean to an integer mask with
    # to values: 0 or 255, then apply the mask
    visMask = (mask * 255).astype("uint8")
    instance = cv2.bitwise_and(roi, roi, mask=visMask)

    # show the extracted ROI, the mask, along with the
    # segmented instance
    cv2.imshow("ROI", roi)
    cv2.imshow("Mask", visMask)
    cv2.imshow("Segmented", instance)
```

```
# now, extract *only* the masked region of the ROI by passing
# in the boolean mask array as our slice condition
roi = roi[mask]

# randomly select a color that will be used to visualize this
# particular instance segmentation then create a transparent
# overlay by blending the randomly selected color with the ROI
color = random.choice(COLORS)
blended = ((0.4 * color) + (0.6 * roi)).astype("uint8")

# store the blended ROI in the original image
clone[startY:endY, startX:endX][mask] = blended
```

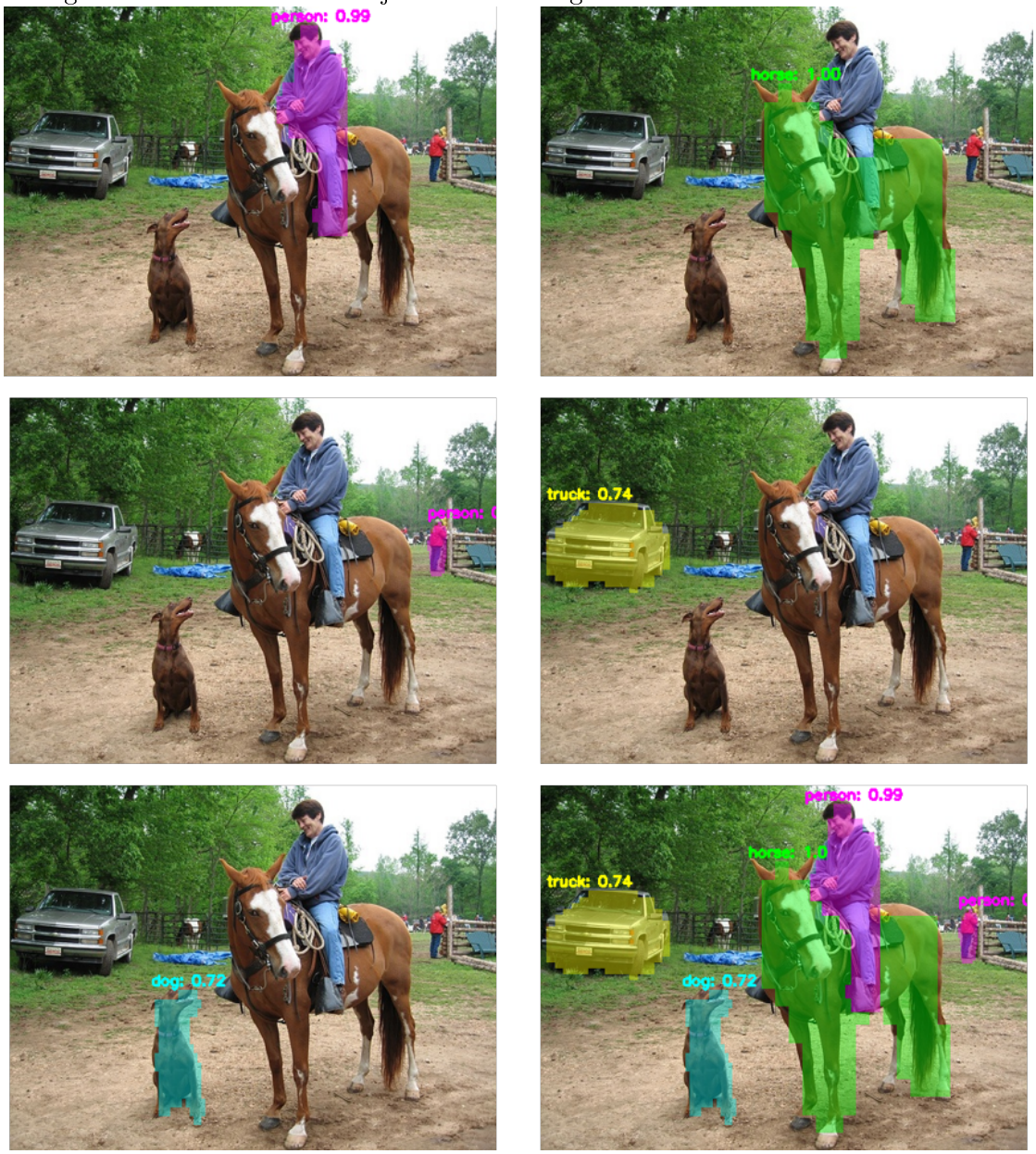
Then after, we extract only the mask region of the ROI by passing the boolean mask array. After that a random color is selected and applied to the transparent overlay. Subsequently, we will blend the masked region with the roi.

```
# draw the bounding box of the instance on the image
color = [int(c) for c in color]
cv2.rectangle(clone, (startX, startY), (endX, endY), color, 2)

# draw the predicted label and associated probability of the
# instance segmentation on the image
text = "{}: {:.4f}".format(LABELS[classID], confidence)
cv2.putText(clone, text, (startX, startY - 5),
    cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

# show the output image
cv2.imshow("Output", clone)
cv2.waitKey(0)
```

IV.2.2 Qualitative Results

The final step was drawing the rectangle and textual class label and confidence value on the image as well as showing the final output image. The following figures show an example of output images.

Figure 11: Results for the object detection algorithm with measures of confidence



IV.3 Image depth computation

IV.3.1 Description

Depth estimation in computer vision and robotics is most commonly done via stereo vision (stereopsis), in which images from two cameras are used to triangulate and estimate distances. However, our project is based on monocular vision. First because the use of a monocular camera reduces the weight and size of robots, and cuts down the cost. Second, although stereo vision systems work well in many environments, they are fundamentally limited by the baseline distance between the two cameras. More specifically, the depth estimates tend to be inaccurate when the distances considered are large. Finally, there are numerous monocular visual cues— such as texture variations and gradients, defocus, color/haze, etc.—that are not exploited in stereo systems.

Depth estimation from monocular vision is a difficult task, because it requires that we take into account the global structure of the image. To address this task, first, we tried the Monodepth algorithm for depth estimation using several street images and based on the cityscapes model. The result was quite satisfactory compared to other models. We then used the improved version of this algorithm (Monodepth 2) which provides a better accuracy, monocular training and shorter training time. Along with this algorithm, we tested two different models for depth estimation: Mono and Mono+stereo, both on the Kitti dataset. For an easier use (color manipulation, speed of execution), we converted the depth image to black and white.

In order to proceed with our experiments, we installed the dependencies PyTorch 0.4.1, TorchVision 0.2.1, Tensorboard 1.4 and opencv 3.3.1. Note that the code for depth estimation could only be run on a single GPU. The script for Monodepth is as follows. First we import the required libraries:

```
from __future__ import absolute_import, division, print_function

import os
import sys
import glob
import argparse
import numpy as np
import PIL.Image as pil

import matplotlib.pyplot as plt

import torch
from torchvision import transforms, datasets

import networks
from layers import disp_to_depth
from utils import download_model_if_doesnt_exist
```

Then the command line arguments are parsed as follows:

```
def parse_args():
    parser = argparse.ArgumentParser(
        description='Simple testing funtion for Monodepthv2 models.')

    parser.add_argument('--image_path', type=str,
                        help='path to a test image or folder of images', required=True)
    parser.add_argument('--model_name', type=str,
                        help='name of a pretrained model to use',
                        choices=[
                            "mono_640x192",
                            "stereo_640x192",
                            "mono+stereo_640x192",
                            "mono_no_pt_640x192",
                            "stereo_no_pt_640x192",
                            "mono+stereo_no_pt_640x192",
                            "mono_1024x320",
                            "stereo_1024x320",
                            "mono+stereo_1024x320"])
    parser.add_argument('--ext', type=str,
                        help='image extension to search for in folder', default="jpg")
    parser.add_argument("--no_cuda",
                        help='if set, disables CUDA',
                        action='store_true')

    return parser.parse_args()
```

The following function is used in order to predict the disparity for a single image or a folder of images.

```
def test_simple(args):
    """Function to predict for a single image or folder of images
    """
    assert args.model_name is not None, \
        "You must specify the --model_name parameter; see README.md for an example"

    if torch.cuda.is_available() and not args.no_cuda:
        device = torch.device("cuda")
    else:
        device = torch.device("cpu")

    download_model_if_doesnt_exist(args.model_name)
    model_path = os.path.join("models", args.model_name)
    print("-> Loading model from ", model_path)
    encoder_path = os.path.join(model_path, "encoder.pth")
    depth_decoder_path = os.path.join(model_path, "depth.pth")

    # LOADING PRETRAINED MODEL
    print("   Loading pretrained encoder")
    encoder = networks.ResnetEncoder(18, False)
    loaded_dict_enc = torch.load(encoder_path, map_location=device)

    # extract the height and width of image that this model was trained with
    feed_height = loaded_dict_enc['height']
    feed_width = loaded_dict_enc['width']
    filtered_dict_enc = {k: v for k, v in loaded_dict_enc.items() if k in encoder.state_dict()}
    encoder.load_state_dict(filtered_dict_enc)
    encoder.to(device)
    encoder.eval()
```

```
    print("   Loading pretrained decoder")
    depth_decoder = networks.DepthDecoder(
        num_ch_enc=encoder.num_ch_enc, scales=range(4))

    loaded_dict = torch.load(depth_decoder_path, map_location=device)
    depth_decoder.load_state_dict(loaded_dict)

    depth_decoder.to(device)
    depth_decoder.eval()

    # FINDING INPUT IMAGES
    if os.path.isfile(args.image_path):
        # Only testing on a single image
        paths = [args.image_path]
        output_directory = os.path.dirname(args.image_path)
    elif os.path.isdir(args.image_path):
        # Searching folder for images
        paths = glob.glob(os.path.join(args.image_path, '*.{}'.format(args.ext)))
        output_directory = args.image_path
    else:
        raise Exception("Can not find args.image_path: {}".format(args.image_path))

    print("-> Predicting on {:d} test images".format(len(paths)))

    # PREDICTING ON EACH IMAGE IN TURN
    with torch.no_grad():
        for idx, image_path in enumerate(paths):

            if image_path.endswith("_disp.jpg"):
                # don't try to predict disparity for a disparity image!
                continue

            # Load image and preprocess
            input_image = pil.open(image_path).convert('RGB')
            original_width, original_height = input_image.size
            input_image = input_image.resize((feed_width, feed_height), pil.LANCZOS)
            input_image = transforms.ToTensor()(input_image).unsqueeze(0)

            # PREDICTION




            # PREDICTION
            input_image = input_image.to(device)
            features = encoder(input_image)
            outputs = depth_decoder(features)

            disp = outputs[("disp", 0)]
            disp_resized = torch.nn.functional.interpolate(
                disp, (original_height, original_width), mode="bilinear", align_corners=False)

            # Saving numpy file
            output_name = os.path.splitext(os.path.basename(image_path))[0]
            name_dest_npy = os.path.join(output_directory, "{}_disp.npy".format(output_name))
            scaled_disp, _ = disp_to_depth(disp, 0.1, 100)
            np.save(name_dest_npy, scaled_disp.cpu().numpy())

            # Saving colormapped depth image
            disp_resized_np = disp_resized.squeeze().cpu().numpy()
            vmax = np.percentile(disp_resized_np, 95)
            name_dest_im = os.path.join(output_directory, "{}_disp.jpg".format(output_name))
            plt.imsave(name_dest_im, disp_resized_np, cmap='magma', vmax=vmax)

            print("   Processed {:d} of {:d} images - saved prediction to {}".format(
                idx + 1, len(paths), name_dest_im))

    print('-> Done!')


if __name__ == '__main__':
    args = parse_args()
    test_simple(args)
```

We then combined object detection using mask r-cnn with depth estimation. In this script of Mask R-CNN, we included the Monodepth algorithm for depth estimation so that we get a disparity image that will be converted to a gray image, and use it to compute the distance estimation.

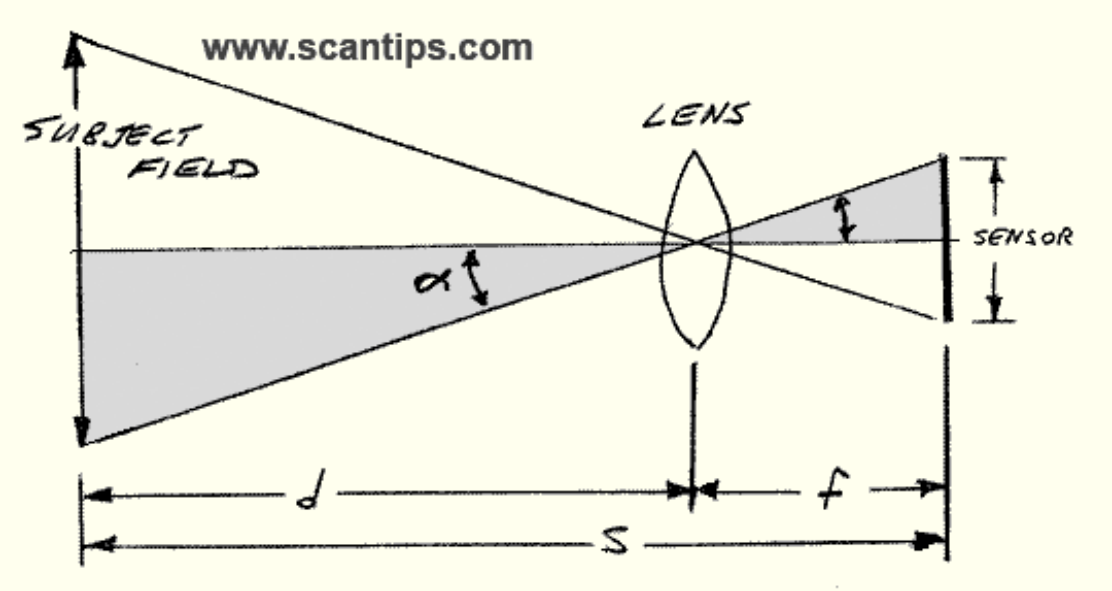
```
# run monodepth
print("### Starting monodepth  #####")
imagename,ext=os.path.splitext(args['image'])
monodepth_models=monodepth_dir+"models/model_"
monodepth_modelname="mono+stereo_640x192"
os.system("python3 "+monodepth_dir+ "test_simple.py --image_path "+args['image']+" --model_name mono+stereo_640x192")
if not os.path.isfile(imagename+"_disp.jpg"):
        print("no monodepth output found")
        sys.exit(-3)
print("### found monodepth output: "+imagename+"_disp.png")
monoimage = cv2.imread(imagename+"_disp.jpg")
monograyimage = cv2.cvtColor(monoimage, cv2.COLOR_BGR2GRAY)

(minVal, maxVal, minLoc, maxLoc) = cv2.minMaxLoc(monograyimage)
print("max="+str(maxVal))
```

The main shortcoming of our approach is that we used a sample of images, for testing the algorithm at this stage, that are not perfectly fit for monodepth training (aerial photographs). Drone images would have been better for the estimation but collecting such data is time-consuming because it requires the use of a real drone, calibration and training of monodepth framework. Consequently, disparity images are not calibrated (since we are not using the same camera), which implies that distance cannot be directly extracted. We therefore had to find a formula to compute distance based on depth estimation. One way of measuring the distance or size of an object or subject in a photo image is using the following Field of View formula

$$distance\ to\ object\ \ (mm) = \frac{focal\ length\ (mm) * real\ height\ of\ the\ object\ (mm) * image\ height\ (pixels)}{object\ height\ (pixels) * sensor\ height\ (mm)} \tag{1}$$

The sensor height is the physical measurement of the vertical dimension of the sensor's active pixel area. The focal length of an optical system is a measure of how strongly the system converges or diverges light; it is the inverse of the system's optical power. The rationale behind this formula is as follows. As shown on the following thin lens model, the field dimension in front of the represented lens node has the same angle (opposite angles) as the sensor angle behind the lens. The focal length corresponds to the distance from the sensor plane to the lens node

Figure 12: Thin Lens model



Therefore we can write

$$\frac{Sensor\ dimension\ \ (mm)}{Focal\ length\ \ (mm)} = \frac{Field\ dimension}{Distance\ to\ field} \tag{2}$$

Similarly for an object

$$\frac{Object\ height\ on\ sensor\ \ (mm)}{Focal\ length\ \ (mm)} = \frac{Real\ Object\ size}{Distance\ to\ Object} \tag{3}$$

Where the object height on sensor is measured as follows

$$Object\ height\ on\ sensor\ \ (mm) = \frac{Sensor\ height\ \ (mm)\ x\ Object\ height\ \ (pixels)}{Sensor\ height\ \ (pixels)} \tag{4}$$

Using the obtained formula (Equation 1) for the distance of object, we estimated the distance for a sample of 20 objects in 7 images, the focal length value was taken from Exif data. Then we estimated the mean of grayscale using Monodepth (scale of color returned from depth estimation) for those same objects. In order to find the formula linking the two variables (distance and mean of grayscale), we proceeded with an Ordinary Least Squares estimation whereby for a model

$$y_i = \sum_{j=1}^{p} X_{ij}\beta_i \tag{5}$$

Table 3: Distance estimation and grayscale value for a sample of objects

Picture	Object	Focal Length	Real height object mm	Image height pixel	Object height pixels	Sensor height mm	Distance in mm	Mean of Grayscale value
A.Center	Car	3.6	1500	3120	951.0	4.69	3 777.4	146.4
Dublin	Pers.1	1.641	1530	1600	139.1	2.98	9 689.1	49.42
Dublin	Car	1.641	1500	1600	139.1	2.98	9 499.1	57.97
Dublin	Pers. 2	1.641	1620	1600	579.7	2.98	2 462.2	119.45
Dublin	Pers. 3	1.641	1580	1600	556.5	2.98	2 501.4	111.64
Parking	Dark car	3.6	1450	3120	785.6	4.69	4 420.2	113.57
Parking	White car	3.6	1470	3120	407.0	4.69	8 650.7	54.31
Highway	Car	3.6	1770	3120	384.3	4.69	11 028.9	47.05
Tower	Pers 3	1.641	1700	1200	156.5	2.98	7 177.1	63.8
Tower	Pers 2	1.641	1680	1200	165.2	2.98	6 719.3	76.44
Tower	Pers 1	1.641	1550	1200	278.3	2.98	3 680.9	143.2
Tower	Pillar	1.641	2000	1200	521.7	2.98	2 533.1	143.2
Holidays	Person	1.641	1500	1200	591.3	2.98	1 676.3	128.54
Holidays	Palm Tree	1.641	7000	1200	173.9	2.98	26 597.4	33.36
Malta	Car 1	1.641	1580	1600	94.9	2.98	14 666.7	172.7
Malta	Car 2	1.641	1520	1600	122.0	2.98	10 974.3	179.6
Malta	Car 3	1.641	1400	1600	122.0	2.98	10 107.9	134.62
Malta	Car 4	1.641	1450	1600	135.6	2.98	9 422.0	133.86
Malta	Car 5	1.641	1500	1600	135.6	2.98	9 746.9	167.33
Malta	Car 6	1.641	1680	1600	216.9	2.98	6 822.8	194.19

Where p is the number of regressors or explanatory variables, X is the matrix containing data on all explanatory variables for the variable y, and i is the number of observations. Coefficients of the explanatory variables are determined through the relation:

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T y \tag{6}$$

As the relationship between distance and mean of grayscale measure appears to be non-lineary from the observations, we tried different transformations of the x variable, corresponding to the mean of grayscale (logarithmic, exponential, power). The following model provided the best fit with a minimized estimation error The P-value indicates

Table 4: Statistical model for distance estimation

Explanatory variables	**Coefficient**	**Standard-Error**	**P-value**
ln(Mean of grayscale)	-5064	2233	0.04
Constant	31489	10374	0.01

Coefficient of determination= explained variance over total variance= 0.47 (Number of observations: 20)

that the coefficients are statistically significant at the 5% level. We therefore decided to use this relation for distance estimation.

$$distance = -5064 * ln(mean\ of\ grayscale) + 31489 \tag{7}$$

This formula was used to adjust the algorithm for point cloud computation based on the estimated depth in the following way: Z = depth.getpixel((u,v))

Z = -5064 * np.log(Z) + 31489

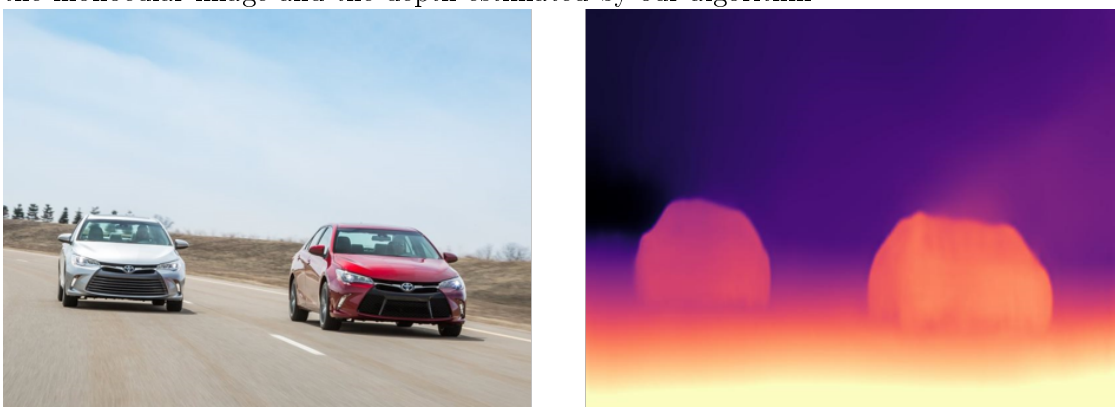
Z = Z / scalingFactor

Where Z is the depth value for a given pixel coordinate (u,v) After a qualitative evaluation of our model based on images from internet for which exif data are available, with different focal lengths, we were able to confirm that our model is a valid representation of reality.

IV.3.2 Qualitative Results

Upon visual inspection of the image samples, we observe that the depth estimation algorithm manages to successfully capture most of the global depth variations (see example below). However, depth estimation was more challenging for images with very distant objects and cluttered backgrounds.

Figure 13: Qualitative results of depth estimation (Monodepth 2). From left to right, the monocular image and the depth estimated by our algorithm



The obtained disparity images were then used for the following step of point cloud computation.

IV.4 Point Cloud computation

IV.4.1 Description

In the following step, we converted the depth estimation to a level of gray representing the distance from the camera (in RGB system a number between 255 white and 0 black) and then used the obtained distance formula (Equation 7) to infer the point cloud coordinates (x, y ,z). The result was then saved in a .ply file with the original image color. The script is as follows:

```
import argparse
import sys
import os
from PIL import Image
import pptk
import numpy as np
import glob
import cv2

focalLength = 1640.0
centerX = 0
centerY = 0
scalingFactor = 500

def generate_pointcloud():#rgb_file="",depth_file="",ply_file=""):
    print("point cloud")
    rgb_folder="/home/youssef/Bureau/monodepth2/assets/testvideo/photos"
    depth_folder="/home/youssef/Bureau/monodepth2/assets/testvideo/dispa"
    ply_file = "/home/youssef/Bureau/monodepth2/assets/testvideo/final.ply"
    rgb = Image.open(rgb_file)

    depth = Image.open(depth_file).convert('I')
    if rgb.size != depth.size:
        raise Exception("Color and depth image do not have the same resolution.")
    if rgb.mode != "RGB":
        raise Exception("Color image is not in RGB format")
    if depth.mode != "I":
        raise Exception("Depth image is not in intensity format")
```

The point cloud coordinates are estimated through the commands

Z = depth.getpixel((u,v))

Z = -5064 * np.log(Z) + 31489

Z = Z / scalingFactor

X = (u - centerX) * Z / focalLength

Y = (v - centerY) * Z / focalLength

We then used Meshlab to visualize the result and calibrate the scaling factor.

```
        points = []
        print(str(rgb.size[0])+" x "+str(rgb.size[1]))
        for v in range(rgb.size[1]):
            for u in range(rgb.size[0]):
                color = rgb.getpixel((u,v))
                Z = depth.getpixel((u,v))
                Z = -5064 * np.log(Z) + 31489
                Z = Z / scalingFactor
                #print(Z)
                #print(u)
                #print(v)
                if Z==0: continue
                X = (u - centerX) * Z / focalLength
                Y = (v - centerY) * Z / focalLength
                points.append("%f %f %f %d %d %d 0\n"%(X,Y,Z,color[0],color[1],color[2]))
            #points.append("%f %f %f %d %d %d 0\n" % (X, Y, Z, color[0], color[1], color[2]))
            #pptk.viewer(points)
        file = open(ply_file, "w")
        file.write('''ply
            format ascii 1.0
            element vertex %d
            property float x
            property float y
            property float z
            property uchar red
            property uchar green
            property uchar blue
            property uchar alpha
            end_header
            %s
            ''' % (len(points), "".join(points)))
        file.close()
        # pptk.viewer(plyf_iole)


    if __name__ == '__main__':
        generate_pointcloud()
        if False:
            parser = argparse.ArgumentParser(description='''
                    This script reads a registered pair of color and depth images and generates a colored 3D point cloud in the
                    PLY format.
                    ''')
            parser.add_argument('rgb_file', help='input color image (format: png)')
            parser.add_argument('depth_file', help='input depth image (format: png)')
            parser.add_argument('ply_file', help='output PLY file (format: ply)')
            args = parser.parse_args()

            generate_pointcloud(args.rgb_file, args.depth_file, args.ply_file)

```

IV.4.2 Qualitative Results

The following figures show qualitative results for an image taken from our sample.

Figure 14: Point cloud estimation, input images

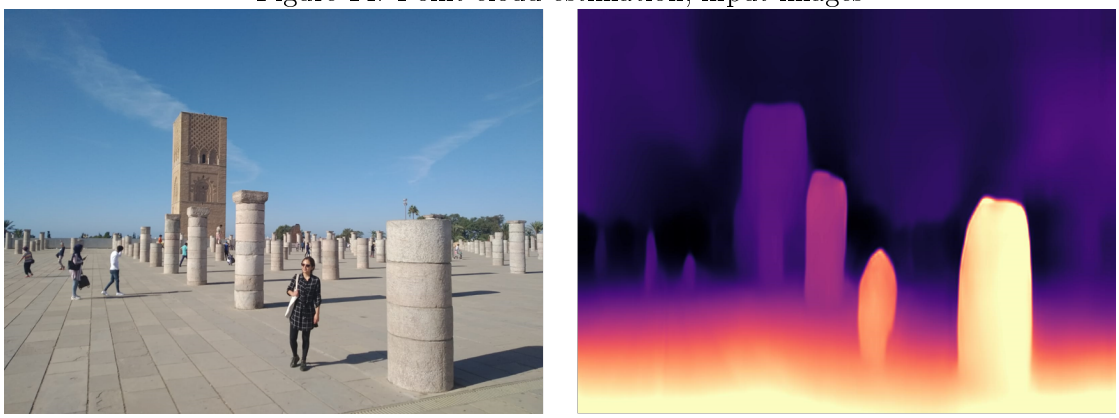
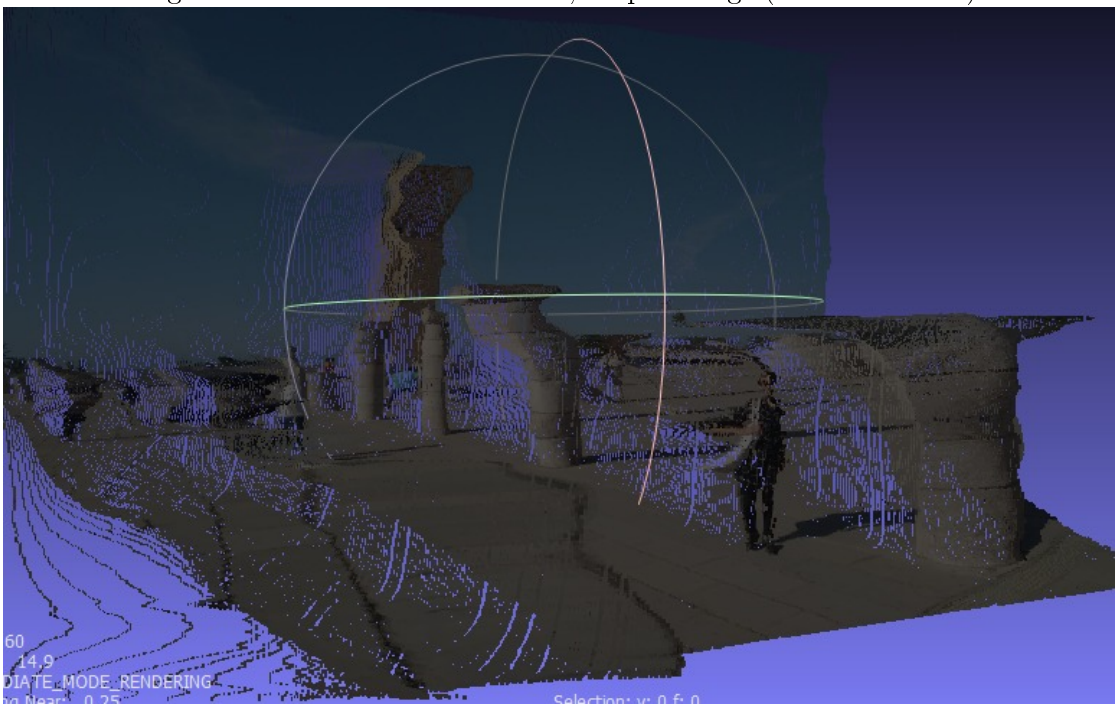


Figure 15: Point cloud estimation, output image (Mesh front view)

Figure 16: Point cloud estimation, output image (Mesh side view)



IV.5 Octomap computation

IV.5.1 Description

In order to build a 3D map with several images, we tried to test different solutions. Softwares such as Lumion, Pix4D Mapper photogrammetry,SimActive Correlator3D^{TM} and others are available. However, in most cases these are costly closed source softwares. We tried the open-source solution OpenDroneMap (`https://www.opendronemap.org/`)but the result was not satisfying due to slow processing. In addition, marking of the entire field is necessary before launching the drone and the resulting map size is too large to be included in the drone. Finally, we chose to work with Octomap which has many advantages such as its small size and its speed.

As stated in the literature review section, the approach of Octomap uses a tree-based representation to offer maximum flexibility with regard to the mapped area and resolution. Its performs a probabilistic occupancy estimation to ensure updatability and to cope with sensor noise. To convert the point clouds to octrees, we first tried to use the Octomap python version. However, we were confronted by a difficulty while trying to install octomap with pip in a conda environment and obtained an import

error. Therefore, we tried to use the C++ version instead. We first proceeded with the conversion of point clouds to pcd through the following script.

```
#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/console/print.h>
#include <pcl/console/parse.h>
#include <pcl/console/time.h>

using namespace pcl;
using namespace pcl::io;
using namespace pcl::console;

void
printHelp (int, char **argv)
{
  print_error ("Syntax is: %s [-format 0|1] input.ply output.pcd\n", argv[0]);
}

bool
loadCloud (const std::string &filename, pcl::PCLPointCloud2 &cloud)
{
  TicToc tt;
  print_highlight ("Loading "); print_value ("%s ", filename.c_str ());

  pcl::PLYReader reader;
  tt.tic ();
  if (reader.read (filename, cloud) < 0)
    return (false);
  print_info ("[done, "); print_value ("%g", tt.toc ()); print_info (" ms : "); print_value ("%d", cloud.width * cloud.height); print_info (" points]\n");
  print_info ("Available dimensions: "); print_value ("%s\n", pcl::getFieldsList (cloud).c_str ());

  return (true);
}

void
saveCloud (const std::string &filename, const pcl::PCLPointCloud2 &cloud, bool format)
{
  TicToc tt;
  tt.tic ();

  print_highlight ("Saving "); print_value ("%s ", filename.c_str ());

  pcl::PCDWriter writer;
  writer.write (filename, cloud, Eigen::Vector4f::Zero (), Eigen::Quaternionf::Identity (), format);

  print_info ("[done, "); print_value ("%g", tt.toc ()); print_info (" ms : "); print_value ("%d", cloud.width * cloud.height); print_info (" points]\n");
}

/* ---[ */
int


 print_info ("Convert a PLY file to PCD format. For more information, use: %s -h\n", argv[0]);

 if (argc < 3)
 {
   printHelp (argc, argv);
   return (-1);
 }

 // Parse the command line arguments for .pcd and .ply files
 std::vector<int> pcd_file_indices = parse_file_extension_argument (argc, argv, ".pcd");
 std::vector<int> ply_file_indices = parse_file_extension_argument (argc, argv, ".ply");
 if (pcd_file_indices.size () != 1 || ply_file_indices.size () != 1)
 {
   print_error ("Need one input PLY file and one output PCD file.\n");
   return (-1);
 }

 // Command line parsing
 bool format = true;
 parse_argument (argc, argv, "-format", format);
 print_info ("PCD output format: "); print_value ("%s\n", (format ? "binary" : "ascii"));

 // Load the first file
 pcl::PCLPointCloud2 cloud;
 if (!loadCloud (argv[ply_file_indices[0]], cloud))
   return (-1);

 // Convert to PLY and save
 saveCloud (argv[pcd_file_indices[0]], cloud, format);

 return (0);
}
```

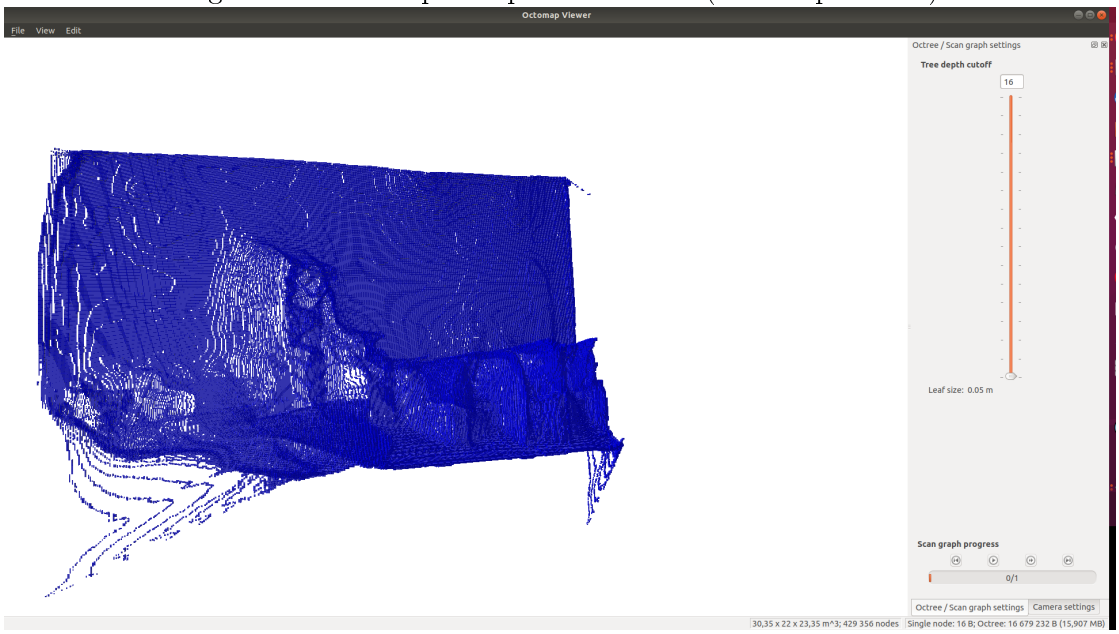
And then we converted the pcd in octrees as follows

```
#include <iostream>
#include <assert.h>

//pcl
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

//octomap
#include <octomap/octomap.h>
using namespace std;

int main( int argc, char** argv )
{
    if (argc != 3)
    {
        cout<<"Usage: pcd2octomap <input_file> <output_file>"<<endl;
        return -1;
    }

    string input_file = argv[1], output_file = argv[2];
    pcl::PointCloud<pcl::PointXYZRGBA> cloud;
    pcl::io::loadPCDFile<pcl::PointXYZRGBA> ( input_file, cloud );

    cout<<"point cloud loaded, piont size = "<<cloud.points.size()<<endl;


    cout<<"copy data into octomap..."<<endl;

    octomap::OcTree tree( 0.05 );

    for (auto p:cloud.points)
    {

        tree.updateNode( octomap::point3d(p.x, p.y, p.z), true );
    }


    tree.updateInnerOccupancy();

    tree.writeBinary( output_file );
    cout<<"done."<<endl;

    return 0;
```

IV.5.2 Qualitative Results

The registered point clouds from the previous images were transformed into the more compact OctoMap representation. An example is provided in the following figure using the OctoMap viewer.

Figure 17: Octomap Computation result (Octomap Viewer)



V Conclusion and future works

The aim of this project was to conceive a solution for situation awareness using monocular cameras. To achieve this goal we combined and adapted several algorithms related to object detection, image segmentation (Mask R-CNN) , depth estimation (Monodepth), point cloud estimation, and Octomap computation.

First we used Mask R-CNN for image segmentation, which gave use a good qualitative result and execution time. Then we included Monodepth for depth estimation, and obtained a good estimation of distance after creating a calibration algorithm based on focus length and tests from different images. The combination of Monodepth for depth estimation and our calibration algorithm, generated an output that we converted into a map. Point cloud visualization and different testings reinforced our confidence in the obtained results, even though the 3d map could have been better with the use of several images. From a qualitative evaluation of the final results we conclude that our solution provides a satisfactory output map with no need for high frame rates, which is very promising for monocular camera-based situational awareness.

Future steps for this project would be the use of the Robot Operating System (ROS) platform, and the Gazebo 3D simulator to test the solution. At this stage, we could use have several images for an object based on different positions of the camera. Other

approaches could be included such as the use of GPS or SLAM. Then it would be possible to represent the detected objects through mask r-cnn as a cuboid with point clouds so that the drone would be able to recognize them as obstacles and avoid them. At this stage, more powerful algorithms such as Visual-Interial Odometry would be helpful.

Bibliography

Borenstein, E. and S. Ullman (2002). Class-specific, top-down segmentation. In *European conference on computer vision*, pp. 109–122. Springer.

Brownlee, J. (2019). A gentle introduction to object recognition with deep learning.

Cao, Y., Z. Wu, and C. Shen (2017). Estimating depth from monocular images as classification using deep fully convolutional residual networks. *IEEE Transactions on Circuits and Systems for Video Technology 28*(11), 3174–3182.

Chaurasia, G., S. Duchene, O. Sorkine-Hornung, and G. Drettakis (2013). Depth synthesis and local warps for plausible image-based navigation. *ACM Transactions on Graphics (TOG) 32*(3), 30.

Chaurasia, G., O. Sorkine, and G. Drettakis (2011). Silhouette-aware warping for image-based rendering. In *Computer Graphics Forum*, Volume 30, pp. 1223–1232. Wiley Online Library.

Cole, D. M. and P. M. Newman (2006). Using laser range data for 3d slam in outdoor environments. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 1556–1563. IEEE.

Cuerno-Rejado, C., L. García-Hernández, A. Sánchez-Carmona, A. Carrio, J. L. Sánchez-López, and P. Campoy (2016). Historical evolution of the unmanned aerial vehicles to the present.

Dai, J., K. He, Y. Li, S. Ren, and J. Sun (2016). Instance-sensitive fully convolutional networks. In *European Conference on Computer Vision*, pp. 534–549. Springer.

Dai, J., K. He, and J. Sun (2015). Convolutional feature masking for joint object and stuff segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3992–4000.

Eigen, D. and R. Fergus (2015). Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, Washington, DC, USA, pp. 2650–2658. IEEE Computer Society.

Eigen, D., C. Puhrsch, and R. Fergus (2014). Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pp. 2366–2374.

Eisemann, M., B. De Decker, M. Magnor, P. Bekaert, E. De Aguiar, N. Ahmed, C. Theobalt, and A. Sellent (2008). Floating textures. In *Computer graphics forum*, Volume 27, pp. 409–418. Wiley Online Library.

Fitzgibbon, A., Y. Wexler, and A. Zisserman (2005). Image-based rendering using image-based priors. *International Journal of Computer Vision 63*(2), 141–151.

Flynn, J., I. Neulander, J. Philbin, and N. Snavely (2016). Deepstereo: Learning to predict new views from the world's imagery. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5515–5524.

Garcia-Garcia, A., S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. Garcia-Rodriguez (2017). A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*.

Garg, R., V. K. BG, G. Carneiro, and I. Reid (2016). Unsupervised cnn for single view depth estimation: Geometry to the rescue. In *European Conference on Computer Vision*, pp. 740–756. Springer.

Girshick, R., J. Donahue, T. Darrell, and J. Malik (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

Godard, C., O. Mac Aodha, and G. J. Brostow (2017). Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 270–279.

Goesele, M., J. Ackermann, S. Fuhrmann, C. Haubold, R. Klowsky, D. Steedly, and R. Szeliski (2010). Ambient point clouds for view interpolation. *ACM Transactions on Graphics (TOG) 29*(4), 95.

Graves, A., S. Fernández, and J. Schmidhuber (2007). Multi-dimensional recurrent neural networks. In *International conference on artificial neural networks*, pp. 549–558. Springer.

Grover, P. (2018). Evolution of object detection and localization algorithms.

Han, F. and S.-C. Zhu (2003). Bayesian reconstruction of 3d shapes and scenes from a single image. In *First IEEE International Workshop on Higher-Level Knowledge in 3D Modeling and Motion Analysis, 2003. HLK 2003.*, pp. 12–20. IEEE.

Hariharan, B., P. Arbeláez, R. Girshick, and J. Malik (2015). Hypercolumns for object segmentation and fine-grained localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 447–456.

Harwood, R. (2019). The challenges to developing fully autonomous drone technology.

Hornung, A., M. Phillips, E. G. Jones, M. Bennewitz, M. Likhachev, and S. Chitta (2012). Navigation in three-dimensional cluttered environments for mobile manipulation. In *2012 IEEE International Conference on Robotics and Automation*, pp. 423–429. IEEE.

Hornung, A., K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous robots 34*(3), 189–206.

Jaderberg, M., K. Simonyan, A. Zisserman, et al. (2015). Spatial transformer networks. In *Advances in neural information processing systems*, pp. 2017–2025.

Karsch, K., C. Liu, and S. B. Kang (2014). Depth transfer: Depth extraction from video using non-parametric sampling. *IEEE transactions on pattern analysis and machine intelligence 36*(11), 2144–2158.

Krizhevsky, A., I. Sutskever, and G. Hinton (2014). Imagenet classification with deep convolutional neural. In *Neural Information Processing Systems*, pp. 1–9.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

Ladicky, L., J. Shi, and M. Pollefeys (2014). Pulling things out of perspective. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 89–96.

Liu, F., C. Shen, G. Lin, and I. Reid (2015). Learning depth from single monocular images using deep convolutional neural fields. *IEEE transactions on pattern analysis and machine intelligence 38*(10), 2024–2039.

Long, J., E. Shelhamer, and T. Darrell (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440.

Nagai, T., T. Naruse, M. Ikehara, and A. Kurematsu (2002). Hmm-based surface reconstruction from single images. In *Proceedings. International Conference on Image Processing*, Volume 2, pp. II–II. IEEE.

Pfaff, P., R. Triebel, and W. Burgard (2007). An efficient extension to elevation maps for outdoor terrain mapping and loop closing. *The International Journal of Robotics Research 26*(2), 217–230.

Pinheiro, P. O., R. Collobert, and P. Dollár (2015). Learning to segment object candidates. In *Advances in Neural Information Processing Systems*, pp. 1990–1998.

Pinheiro, P. O., T.-Y. Lin, R. Collobert, and P. Dollár (2016). Learning to refine object segments. In *European Conference on Computer Vision*, pp. 75–91. Springer.

Redmon, J. and A. Farhadi (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

Roy, A. and S. Todorovic (2016). Monocular depth estimation using neural regression forest. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5506–5514.

Ryde, J. and H. Hu (2010). 3d mapping with multi-resolution occupied voxel lists. *Autonomous Robots 28*(2), 169.

Saxena, A., S. H. Chung, and A. Y. Ng (2006). Learning depth from single monocular images. In *Advances in neural information processing systems*, pp. 1161–1168.

Saxena, A., S. H. Chung, and A. Y. Ng (2008). 3-d depth reconstruction from a single still image. *International journal of computer vision 76*(1), 53–69.

Saxena, A., M. Sun, and A. Y. Ng (2009). Make3d: Learning 3d scene structure from a single still image. *IEEE transactions on pattern analysis and machine intelligence 31*(5), 824–840.

Sermanet, P., D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*.

Simonyan, K. and A. Zisserman (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Visin, F., K. Kastner, K. Cho, M. Matteucci, A. Courville, and Y. Bengio (2015). Renet: A recurrent neural network based alternative to convolutional networks. *arXiv preprint arXiv:1505.00393*.

Wang, X., D. Fouhey, and A. Gupta (2015). Designing deep networks for surface normal estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 539–547.

Xie, J., R. Girshick, and A. Farhadi (2016). Deep3d: Fully automatic 2d-to-3d video conversion with deep convolutional neural networks. In *European Conference on Computer Vision*, pp. 842–857. Springer.

Zagoruyko, S., A. Lerer, T.-Y. Lin, P. O. Pinheiro, S. Gross, S. Chintala, and P. Dollár (2016). A multipath network for object detection. *arXiv preprint arXiv:1604.02135*.

Zhang, Z., S. Fidler, and R. Urtasun (2016). Instance-level segmentation for autonomous driving with deep densely connected mrfs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 669–677.

Zhang, Z., A. G. Schwing, S. Fidler, and R. Urtasun (2015). Monocular object instance segmentation and depth ordering with cnns. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2614–2622.

Zhao, Z.-Q., P. Zheng, S.-t. Xu, and X. Wu (2019). Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems 30*(11), 3212–3232.

List of Tables

1 Comparison between YOLOv3 and other detectors, in terms of inference time and mAP Redmon and Farhadi (2018) 18
2 Deep learning semantic segmentation methods 19
3 Distance estimation and grayscale value for a sample of objects 41
4 Statistical model for distance estimation 42

List of Figures

1 Overview of Object Recognition Computer Vision Tasks (Brownlee (2019)) 10

2 VGG16 12

3 The R-CNN system 14

4 Fast R-CNN 15

5 Faster R-CNN 16

6 Network architecture (YOLO) 17

7 Bounding-box predictions (YOLO) 17

8 Difference between object recognition tasks Garcia-Garcia et al. (2017) . . 20

9 A typical elevation map for a bridgePfaff et al. (2007) 26

10 Example of an octree storing free shaded white and occupied black cells (from left to right: the volumetric model and the tree representation), Hornung et al. (2013) 27

11 Results for the object detection algorithm with measures of confidence . . 34

12 Thin Lens model 40

13 Qualitative results of depth estimation (Monodepth 2). From left to right, the monocular image and the depth estimated by our algorithm 43

14 Point cloud estimation, input images 46

15 Point cloud estimation, output image (Mesh front view) 46

16 Point cloud estimation, output image (Mesh side view) 47

17 Octomap Computation result (Octomap Viewer) 50