NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

A Leader-Follower Mobile Robot Scheme using an RGB-D Camera and MobileNets

Panagiotis A. Petropoulakis

Supervisor: **Ioannis Emiris,** Professor

ATHENS

October 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχήμα Οδηγός-Ακόλουθος Κινούμενων Ρομπότ με την χρήση RGB-D Κάμερας και MobileNets

Παναγιώτης Α. Πετροπουλάκης

Επιβλέπων: **Ιωάννης Εμίρης,** Καθηγητής

ΑΘΗΝΑ

Οκτώβριος 2020

BSc THESIS

A Leader-Follower Mobile Robot Scheme using an RGB-D Camera and MobileNets

Panagiotis A. Petropoulakis

S.N.: 1115201500129

SUPERVISOR: **Ioannis Emiris,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχήμα Οδηγός-Ακόλουθος Κινούμενων Ρομπότ με την χρήση RGB-D Κάμερας και MobileNets

Παναγιώτης Α. Πετροπουλάκης

Α.Μ.: 1115201500129

ΕΠΙΒΛΕΠΩΝ: Ιωάννης Εμίρης, Καθηγητής

ABSTRACT

Robotic systems that operate in unstructured environments are mainly relying on sense and perception. To effectively observe and understand the world, Artificial Intelligence provides novel and robust techniques such as object detection. In this work, we examine the performance of MobileNets[1] detection system in a Leader-Follower Mobile Robot Scheme using an RGB-D Camera. We additionally gather real examples to train the model and made the dataset available online for potential research. Multiple experiments on Pioneer 3-AT and Summit-XL robots show that MobileNets offer an exceptional detection accuracy. The proposed Leader-Follower Scheme runs at 19 Hz on GeForce GTX1060 GPU card with less than 5% of false detections. The source code for the complete system can be found at: https://github.com/PetropoulakisPanagiotis/BSc_thesis.

SUBJECT AREA: Robotics

KEYWORDS: Robotics, Visual-Servoing, Deep Learning, Leader-Follower Scheme, RGB-D Camera

ΠΕΡΙΛΗΨΗ

Τα Ρομποτικά Συστήματα που ενεργούν σε αδόμητα περιβάλλοντα βασίζονται κυρίως στην όραση και στις αισθήσεις τους. Για να μπορέσουν να αποτυπώσουν και να καταλάβουν τον περιβάλλοντα χώρο αποδοτικά, η Τεχνιτή Νοημοσύνη προσφέρει σε αυτά καινοτόμες αλλά και ανθετικές μεθόδους, όπως ο εντοπισμός των αντικειμένων. Στην παρούσα εργασία εξετάζουμε την απόδοση ενός τέτοιου συστήματος εντοπισμού που ονομάζεται MobileNets[1] σε ένα σχήμα Οδηγός-Ακόλουθος Κινούμενων Ρομπότ με την χρήση RGB-D Κάμερας. Επιπλέον, μαζεύουμε πραγματικά δεδομένα για να εκπαιδεύσουμε το μοντέλο, τα οποία τα διαθέτουμε προς ελεύθερη χρήση. Πολλαπλά πειράματα στα Pioneer 3-AT και Summit-XL ρομπότ έδειξαν ότι τα MobileNets προσφέρουν εξαιρετική ακρίβεια. Το σύστημα Οδηγός-Ακόλουθος που προτείνουμε τρέχει στα 19 Hz στην GeForce GTX1060 GPU κάρτα με λιγότερο από 5% λανθασμένες προβλέψεις. Ο πηγαίος κώδικας του συστήματος βρίσκετε στον σύνδεσμο: https://github.com/PetropoulakisPanagiotis/BSc_thesis.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ρομποτική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ρομποτική, Οπτικός Έλεγχος, Βαθιά Μάθηση, Σχήμα Οδηγός-Ακόλουθος, RGB-D Κάμερα

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

First of all, my thesis would be impossible to be completed without the contribution of Prof. Kostas J. Kyriakopoulos from the Mech. Eng. Department of National Technical University of Athens. Kyriakopoulos not only gave me the opportunity to be a member of his lab and use his resources like expensive robots, but he guided me one and a half years through my journey to Robotics and Artificial Intelligence. Prior meeting Kyriakopoulos, I hadn't found a field of interest and none of the Robotics and AI concepts were familiar to me, but only on a high level of understanding. During our meetings he gave me valuable advices and pushed me to work hard. In the first place, I came across with scientific research papers and even, in the last months, I managed to deliver my first review on the MED'2020 Control and Automation conference.

Dr. George Karras had also an immense contribution to the present thesis by sharing his knowledge and expertise in visual-servoing.

I thank Michalis Logothetis and PhD student Kostas Alevizos for giving me valuable tips for the final experiments with the mobile robots.

PhD student Sotiris Aspragkathos allotted his valuable time to gather together with me real training data.

Last but not least, I would like to thank a former student from my university, Dr. Marios Xanthidis. I meet him a few years ago, during his BSc thesis presentation and he advised me how to start with Robotics by introducing to me Prof. Kostas J. Kyriakopoulos.

1. INTRODUCTION

Robotic systems are already present in many aspects of our everyday life. Robotic arms perform heavy tasks in factories, drones gradually deliver products to our doors and even, mobile robots collaborate in search and rescue scenarios(refer also to figure 1.1).

Figure 1.1: Different types of robots

Most of these complex systems, though, rely mainly on visual sensors to understand their surrounding environment and as a result, in robotics, the Vision-Based Control[2, 3] area is evolving exponentially.

To be more precise, Vision-Based Control refers to the use of vision data to control the motion of a robot. The vision data may be acquired from a camera that is often mounted on a robot or the so-called eye-in-hand case. This area primarily relies on techniques from computer vision. Some major breakthroughs in various image recognition tasks such as object detection, image segmentation and image reconstruction, have contributed by solving many robotic perception challenges.

In our turn, we study a Leader-Follower Scheme using an RGB-D Camera. We believe that cooperative vehicles will be used in many applications and in the future we will observe more such Schemes.

Figure 1.2: A Leader-Follower Scheme

To begin with, a Leader-Follower Scheme as captured in figure 1.2, consists mainly of two robots and one camera. The end goal here is that, the Leader must remain constantly in the field of view of the Follower.

To accomplish such a task, at first, the Follower should somehow detect the Leader in the image or the RGB frame. Classical detection methods like markers or SURF[4] or even ORB[5] features have gradually faded from real world applications, since autonomy requires robust techniques to tackle with extreme lighting conditions. The appearance of Deep Learning techniques, though, made it possible to go near to real life autonomy.

As a result, we employ MobileNets[1] in our detection system and as we will see later, we are able, with this method, to use a low-end GPU card and achieve excellent performance.

The next challenges in a Leader-Follower Scheme is to acquire the Leader's position with respect to the Follower and then, use this information to steer the Follower towards the Leader. In our Scheme, we retrieve the Leader's position by combining the Depth with the RGB frames of the camera and then, plug a position based controller to steer the Follower.

To sum up, our work is divided into 5 chapters as follows:

1. In chapter 2, we perform a survey on various object detection techniques and architectures to justify the selection of MobileNets on a low-end GPU card.
2. In chapter 3, we discuss in detail the basic concepts, the main components as well as the software design and the network architecture of our Leader-Follower Scheme.
3. In chapter 4, we present our solution to this complex Scheme.
4. In chapter 5, we validate our method in multiple experiments with real mobile robots.
5. Finally, in chapter 6 we present the concluding remarks, our final thoughts and we even talk about possible extensions and the future of the cooperative vehicles.

2. LITERATURE OVERVIEW

In this chapter, we discuss and compare different Object Detection techniques and Neural Network architectures. We begin with Region Proposals methods: R-CNN[6] and its variants[7, 8]. Next, we present real-time Object Detection systems: YOLO[9], SSD[10] and YOLO9000[11]. We continue with ResNet[12], an important learning framework to ease the training of Deep Networks and finish up with light weight models for embedded vision applications, employed in our Leader-Follower Scheme, the MobileNets[1, 13].

2.1 R-CNN: Rich feature hierarchies for accurate object detection and semantic segmentation

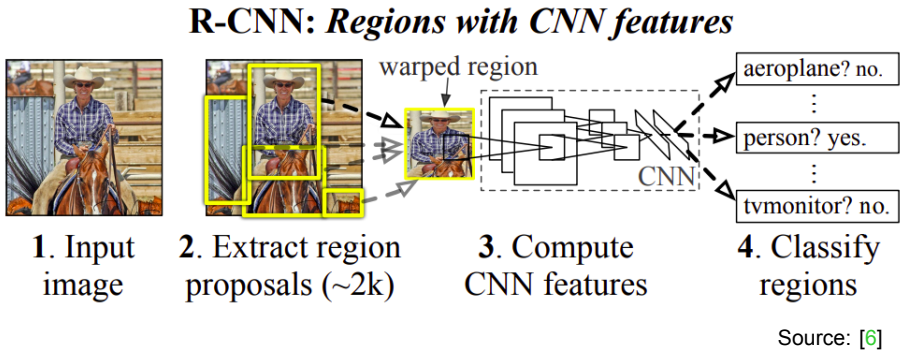


Figure 2.1: R-CNN system overview

At the time, the progress on the Object Detection task was slow, but with the onset of the proposed R-CNN[6] algorithm, the mean average precision(mAP) increased more than 30% relative to the previous best result on VOC2012[14] challenge. In addition, the authors suggested and validated a way to train models effectively with scarce data by the "supervised pre-training/domain-specific fine-tuning" paradigm.

Their final system is divided into three modules as follows(figure 2.1):

1. In the first module, selective search[15] algorithm produces 2000 class-independent region proposals. Initially, the proposals are generated with a hierarchical algorithm and then, the similar regions are combined to larger ones using color, texture, size and shape heuristics. These final proposals form a set of candidates boxes for the detector.

2. In the second module, every proposal forward passes through a convolutional neural network. The architecture is inspired by AlexNet[16] that entails fc, dropout, max pooling and convolutional layers. Though, to be consistent with the original AlexNet's input requirements, each region proposal is resized to a 227×227 pixel size with the affine image wrapping transformation. Eventually, this module produces for every proposal a 4096-dimensional feature vector.

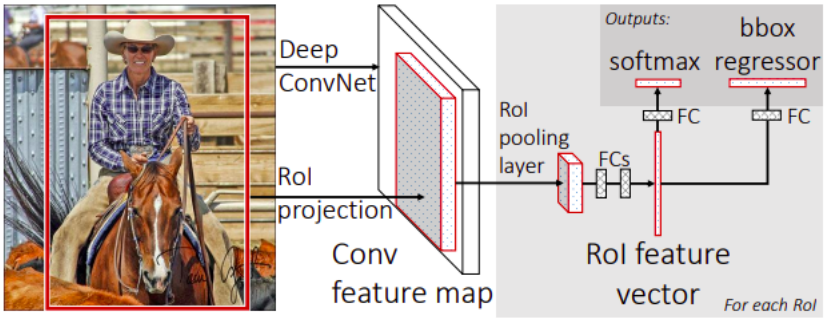
3. Finally, in the last module, each 4096-dim feature vector is classified by a set of class-specific linear SVMs. Additionally, to improve the localization accuracy, one extra step is recommended: a bounding-box regression by using the pool_5 features and the corresponding region proposal coordinates to predict an improved bounding box.

R. Girshick et al.[6] pre-trained the original AlexNet on ILSVRC2012[17] dataset and then fine-tuned it on the warped region proposals by replacing the 1000-way with a (N+1)-way classification layer. Afterwards, they removed the softmax layer and added one linear SVM per class. However, during this phase, some objects are partially overlapped(regions may contain a fraction of an object) and to train effectively these binary classifiers, they defined regions with IoU less than 0.3 as negative examples. Moreover, the authors employed hard negative mining[18] to pick the right amount of negative examples.

This architecture might seem odd to the average reader, but without fine-tuning linear class-specific SVMs the performance on VOC2007 dropped from 54.2% to 50.9%. The softmax layer doesn't offer accurate localization, as it has been trained mainly on randomly sampled negative examples. On the contrary, SVMs were trained on the subset of "hard negatives".

At last, the authors conducted an additional study regarding the layers' performance with or without fine-tuning. They found out that without fine-tuning the performance decreases by having two fc layers. On the hand, fine-tuning on PASCAL enhanced the mAP between the fc_4 and fc_5 layers. Interestingly enough, the mAP of the last $pool_5$ layer wasn't increased significantly. $Pool_5$ learned representation is more general and the overall improvement is gained from learning domain-specific classifiers.

2.2 Fast R-CNN: Fast Rich feature hierarchies for accurate object detection and semantic segmentation



Source: [7]

Figure 2.2: Fast R-CNN architecture

R-CNN[6] was an expensive multi-stage algorithm in which the model is initially fine-tuned with a log loss, then the last layer is replaced with SVMs and finally, a box regressor needs tuning to improve the localization accuracy. For these reasons, R-CNN was a time consuming process and very inefficient memory wise, since the $pool_5$ features must be written to disk to make the final predictions(with the box-regressor). Not only does this procedure take place 2000 times per image, for each proposal, but the regions' features maps do not share any common information.

In this work, Ross Girshick[7] tries to cure all of these problems by proposing a single-stage algorithm that jointly learns to classify object proposals and refine their spatial locations. With this modification, all network weights are updated during the back-propagation and as a result, Fast R-CNN[7] can be trained 9× faster than R-CNN while being 213× faster at test-time.

In Fast R-CNN the whole image is processed by a ConvNet that predicts(only once!) a feature map. Then, each region proposal is mapped/aligned with the corresponding area of the previous map and by using RoI pooling, a max pooling operation, this area is down-sampled to a fixed-size feature map. Finally, the fixed map is forward passes through fc layers and branch into two sibling output layers: one that predicts softmax probability estimates over K+1 object classes and another that predicts four numbers for each of the K object classes to refine the bonding box(figure 2.2). In this way, big region proposals are converted to time-efficient fixed-size maps(with RoI pooling) that share information from the initial feature map.

The strength of Fast R-CNN originates from a multi-task loss. One output layer estimates a discrete probability distribution $p = (p_0, ..., p_K)$ with a softmax layer, for all the $K + 1$ categories, and the second one estimates offsets $t^k = (t^k_x, t^k_y, t^k_w, t^k_h)$ relative to an object proposal, for each class k.

The definition of such a multi-task loss L is the following:

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda[u \geq 1]L_{loc}(t^u, v), \tag{1}$$

in which $L_{cls}(p, u) = -\log(p_u)$ is a log loss for the true class u and λ is a hyper-parameter to attach more or less weight to each task.

The L_{loc} consists of the true $v^u = (v^u_x, v^u_y, v^u_w, v^u_h)$ and the predicted $t^u = (t^u_x, t^u_y, t^u_w, t^u_h)$ bounding-box offsets, for each class u. The Iverson bracket indicator function $[u \geq 1]$ equals to 1 when $u \geq 1$ and 0 otherwise. By convention, the background class is set to $u = 0$.

To be more precise, localization loss is defined as:

$$L_{loc}(t^u, v) = \sum_{i \in \{x,y,w,h\}} smooth_{L_1}(t^u_i - u_i), \tag{2}$$

in which

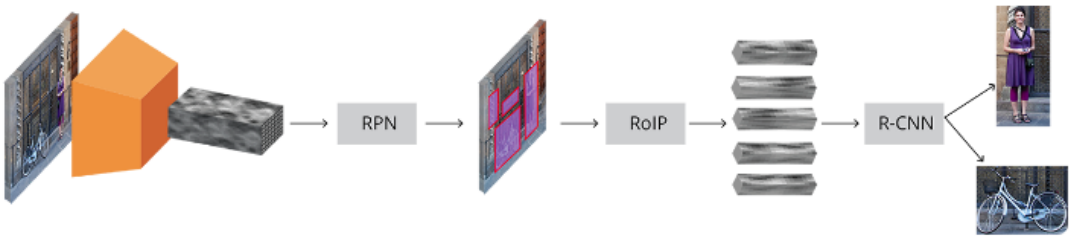
$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & if|x| < 1 \\ |x| - 0.5 & otherwise, \end{cases} \tag{3}$$

is a combination of L_1-loss and L_2-loss. It has less oscillations during updates when x is small and it prevents exploding gradients when x is large.

Final remarks about Fast R-CNN:

1. Mini-batch sampling: each training batch contains N images and R regions(using hierarchical sampling). By setting $N = 2$ and $R = 128$, the training time is 64× faster than $N = 128$ and $R = 2$, since in the latter situation 128 images forward pass through the ConvNet.
2. Ross Girshick[7] validated that multi-task training influences each individual task and the overall performance is increasing.
3. In deeper networks, like VGG-16[19], is suggested to fine-tune the conv layers.
4. Regarding softmax vs SVMs: softmax outperforms, as there is a competition among the classes.
5. Fully connected layers can be accelerated by compressing them with truncated SVD[20, 21] and in this way, the detection time can be reduced more than 30% with only 0.3% drop on mAP.

2.3 Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks



Source: tryolabs article

Figure 2.3: Faster R-CNN architecture

Selective Search[15] was a heavy computational bottleneck in the previous work and Faster R-CNN[8] introduced the novel Region Proposal Network(RPN), a convolutional trained end-to-end network that shares features with the detection network(Fast R-CNN[7]) by using the same backbone. Such a network, predicts state-of-the-art object proposals requiring only 10ms per image.

2.3.1 Network Overview

The input image forward passes through a backbone pre-trained convolutional network and then, the predicted feature map is processed by the RPN that generates bounding boxes with a corresponding ”objectness” probability score(object vs not object). Lastly, by employing the RoI pooling and the final layers of Fast R-CNN, we can estimate class probabilities and refined box offsets. The network’s overall architecture is summarized in figure 2.3.

2.3.2 Anchor Boxes

Source: tryolabs article

Figure 2.4: Anchor centers in Faster R-CNN

The main question here is how to predict object proposals with a fully convolutional neural network. One approach is to generate directly box coordinates $(x_{min}, y_{min}, x_{max}, y_{max})$. Unfortunately, defining such a network is difficult due to some constrains(e.g. unknown number of target boxes and $x_{min} < x_{max}$).

To solve this issue, Faster R-CNN[8] introduced the novel ”anchor” boxes that are fixed reference boxes with predefined scales and aspect-ratios. These anchors defined by $(x_{center}, y_{center}, width, height)$ tuples and then, the RPN simply learns to predict offsets $(\Delta_{x_{center}}, \Delta_{y_{center}}, \Delta_{width}, \Delta_{height})$ relative to the predefined boxes.

To achieve such a behavior, RPN processes a feature map with one conv and two sibling $1{\times}1$ output layers and predicts offsets and scores. More precisely, the first $1{\times}1$ conv layer has an $(1, 1, 2k)$ output shape(per location) representing the ”objectness” score and the second $1{\times}1$ conv layer with $(1, 1, 4k)$ output shape is responsible for the offsets. In this way, k different regressors are learned, one per anchor box for each spatial location.

Imagine having a feature map of $(60{\times}40{\times}9)$ shape(before applying the two $1{\times}1$ layers). The whole information of the input image resides in the depth channels and with $1{\times}1{\times}2k$ and $1{\times}1{\times}4k$ convolutions per location, RPN predicts $60{\times}40{\times}2k$ ”objectness” scores and $60{\times}40{\times}4k$ offsets. Given that the feature map is generated only by using conv and pool layers, one can easily transform every predicted offset and anchor box coordinates with respect to the input image(the predicted offsets and anchors are defined relative to the feature map). For instance, the so-called sub-sampling ratio of an $(1000{\times}600)$ input image and a $(60{\times}40{\times}9)$ feature map is: $\lfloor \frac{1000}{60} \rfloor = 16$. A box in the input image with $(320, 128)$ size and $(340, 450)$ center coordinates has $(20, 8)$ size and $(21, 28)$ center coordinates with respect to the feature map. Keep in mind, that each anchor box center is separated by 16 pixels in the input image(similarly to figure 2.4) and it is rational to choose carefully the number of anchors and the sub-sampling ratio, so as to cover the whole input image. The authors suggest to use 9 anchors per point with $(128, 256, 512)$ size and $(2{:}1, 1{:}1, 1{:}2)$ aspect ratios, respectively.

Finally yet importantly, this definition of anchors is Translation-Invariant. If an object is translated in an image, then the proposal should translate and the same function can predict the proposal in either location.

2.3.3 Train RPN

One can achieve the aforementioned behavior by minimizing the following multi-task loss:

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*), \quad (1)$$

where i is the index of an anchor and p_i is the corresponding "objectness" score. p_i^* activates the regression part provided that the anchor i is positive or $p_i^* = 1$. The t_i is a tuple of the predicted offsets and t_i^* is the true tuple. The L_{cls} is a log loss and L_{reg} has the same form as defined in Fast R-CNN[7].

Anchors are labeled as positive examples:

1. by having an Intersection-over-Union(IoU) overlap higher than 0.7 or
2. by having the highest IoU overlap with a ground-truth box, respectively.

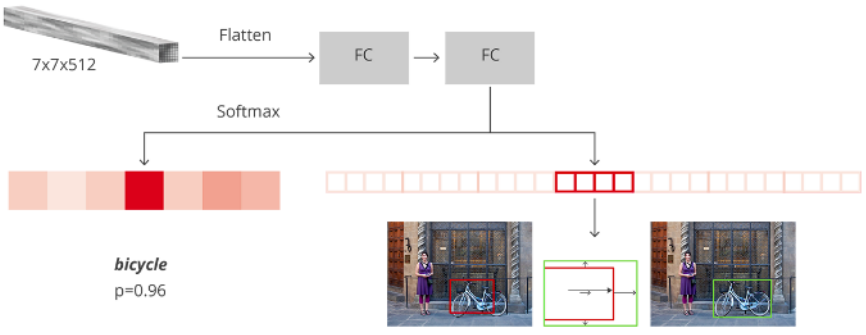
A negative label is assigned to a non-positive anchor box when it has an IoU less than 0.3 for all the ground-truth boxes. The authors suggest to use $N_{cls} = 256$ random samples of anchors per image with a positive-negative ratio of $1{:}1$.

Inevitability, some proposals will highly overlap with each other and to lower the excessive amount of predictions non-maximum suppression(NMS) is employed which is based on their cls scores and on threshold of 0.7 IoU. This leaves us with about 2000 proposals per image, but we can reduce even further the predictions by keeping the top-N. Even with the top-300 predictions, Faster R-CNN offers an exceptional localization accuracy.

One last issue is that some anchors might be cross-boundary. During the training time they are ignored and at test time they are clipped to the image boundary.

2.3.4 Putting it all together

The last stage of Faster R-CNN is based on the previous work of Fast R-CNN: every single proposal must be classified into one class and refined by a set of box regressors.



Source: tryolabs article

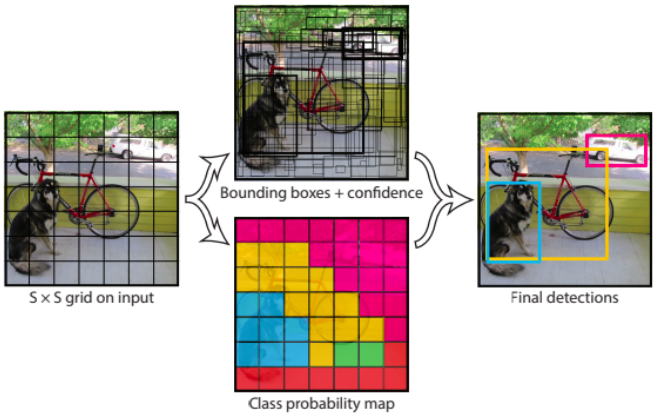
Figure 2.5: Fast R-CNN component inside Faster R-CNN

Each proposal is down-sampled with RoI pooling, one flatten and two fc layers of size 4096 with ReLU. Finally, one fc layer predicts $N + 1$ outputs of class probabilities and another one predicts $4 * N$ offsets $(\Delta_{center_x}, \Delta_{center_y}, \Delta_{width}, \Delta_{height})$ relative to the proposals(figure 2.5). The proposals that have IoU greater than 0.5 with any ground truth box are labeled as positive examples and those that have between 0.1 and 0.5 are labeled as background examples. Like RPN, non-maximum suppression(NMS) post-processing is employed to refine the final predictions. Eventually, by merging RPN with "Fast R-CNN" we end up with four different losses. One can train the whole network end-to-end with a weighted sum over these losses.

2.4 You Only Look Once: Unified, Real-Time Object Detection

R-CNN models[6, 7, 8] are too slow for real-time object detection, because they are multi-stage pipelines. On the contrary, YOLO[9] describes the object detection task as a single regression problem by predicting boxes and class probabilities with one forward pass while running at 45 fps with quite high accuracy. In addition, the network reasons the whole image globally rather than using region proposals. In this way, the number of false positive background examples reduced significantly and the generalization ability on artwork is impressive.

The CNN architecture entails 24 conv layers with 3×3 and $\times1$ kernels, 2 fc, max pooling and dropout layers. In-between the conv layers, the authors add Leaky ReLU as an activation function and in the final layer one linear.

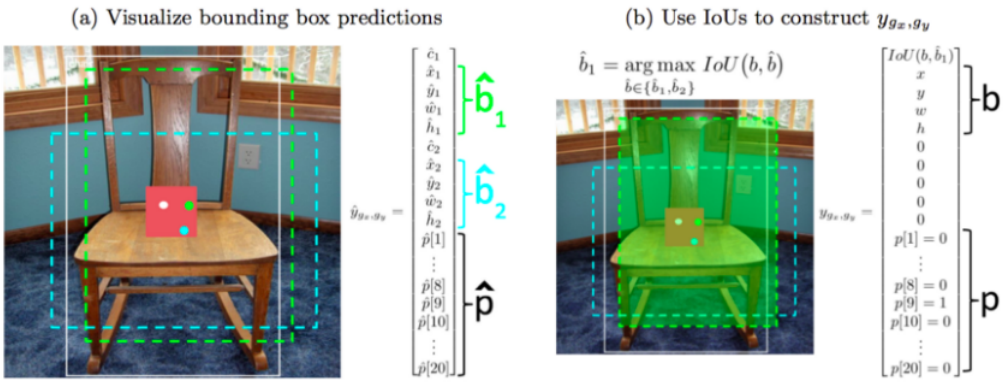


Source: [9]

Figure 2.6: YOLO model

In YOLO, the input image is initially transformed to a square size and then is divided into a $S\times S$ grid. Each of these cells is assigned to B bounding boxes, defined by the following normalized values: $(x_{center}, y_{center}, width, height)$, in which the (x, y) coordinates are relative to the cell and $(width, height)$ are relative to the input image. Additionally, each bounding box is accompanied by a score reflecting the confidence that the box contains an object and also how accurate are the predicted coordinates. The confidence defined as: $Pr(Object) * IOU^{truth}_{pred}$, meaning that the confidence score equals to the intersection over union(IoU) between the predicted box and the ground truth. Finally, each cell predicts C conditional probabilities for all of the B boxes: $Pr(Class_i|Object)$. Authors adopted $S = 7$ and $B = 2$ on PASCAL[14] dataset with $C = 20$(figure 2.6) and the final prediction is a $7\times7\times(2 * 5 + 20)$ tensor.

In the paper it is highlighted that the penalty of small deviations in smaller images must not be equal to the penalty of the larger ones. Joseph Redmon et al.[9] transform the coordinates $(width, height)$ of each B box to $(\sqrt{width}, \sqrt{height})$. However, the problem isn't solved completely and with the definition of one class per cell, YOLO struggles to detect small objects, especially in big groups.



Source: Carol Hsin's medium article

Figure 2.7: Predictions of YOLO

Finally, the CNN "tries" to optimize the following multi-part loss:

$$\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&\quad + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&\quad + \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \sum_{i=0}^{S^2} 1_{i}^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c) \right)^2 , \quad (1)
\end{aligned}$$

where 1_{ij}^{obj} activates the loss when an object appears in cell i with the j-th box being responsible for that detection. Additionally, the authors recommend $\lambda_{coord} = 5$, $\lambda_{noobj} = 0.5$. This selection is due to the fact that a grid cell in the image might not contain objects and then, the confidence score will be zero while the gradients of cells with objects will be high.

A box is assigned to be "responsible" for predicting an object based on which of the $B = 2$ predictions has the highest IOU with the ground truth box, as shown in figure 2.8. At test time, the box(in each cell) with the highest confidence is being kept but the use of NMS can't be omitted. In some cases an object might be quite large or near the border of multiple cells. The final predictions are depicted in figure 2.7.

$$b = \begin{bmatrix} x \\ y \\ w \\ h \end{bmatrix} \quad \hat{b}_1 = \begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \\ \hat{w}_1 \\ \hat{h}_1 \end{bmatrix} \quad \hat{b}_2 = \begin{bmatrix} \hat{x}_2 \\ \hat{y}_2 \\ \hat{w}_2 \\ \hat{h}_2 \end{bmatrix} \quad c = \max_{\hat{b}\in\{\hat{b}_1,\hat{b}_2\}} IoU(b,\hat{b})$$

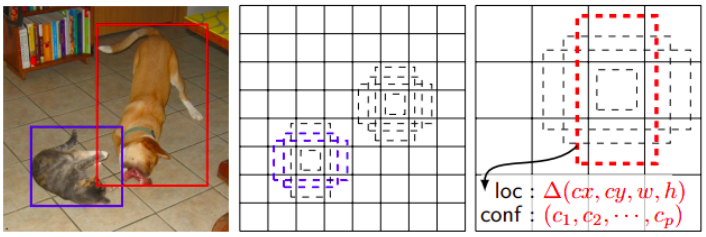
Source: Carol Hsin's medium article

Figure 2.8: Responsible predictor in YOLO

2.5 SSD: Single Shot MultiBox Detector

Henceforth, the Academia turned their attention to real-time but accurate object detection systems. In this work, the Single Shot MultiBox Detector(SDD)[10] beat the benchmarks while being the fastest model. The network generates end-to-end offsets relative to fixed bounding boxes with class probabilities by combining multiple feature maps of different resolutions to improve the detection of objects with different sizes. For each feature map, Wei Liu et al.[10] use small convolutional filters for predictions, but these filters predict box offsets for a specific scale. In this way, SSD achieves state-of-the-art performance of 74.3% mAP at 59 fps on VOC2007[14] with a small input image of 300×300 size. At 512×512 input size, the model has 76.9% mAP and outperforms the Faster R-CNN[8].

2.5.1 Network Overview



Source: [10]

Figure 2.9: Multi-scale feature maps

SSD uses a truncated VGG-16[19](with additional convolutions) as backbone to generate the feature maps(their size is decreasing as we go deeper). Then, a map of $m\times n\times d$ size is convoluted with $(c+4)k$ filters of $3\times3\times d$ size and finally, an output of $m\times n\times d\times k(c+4)$ size is produced. The k here is the number of boxes predicted per cell and $(c+4)$ are the c classes and 4 box offsets. As we mentioned in the introduction, each map is assigned to default boxes with fixed scale and k different aspect ratios. As we can see in figure 2.9, two default boxes are matched with the cat and one with the dog. The 8×8 map predicts relatively small objects and none of the detections can locate the dog. In general, lower resolution feature maps detect larger scale objects.

The final predictions of SSD are made with the following feature maps:

1. The Conv4_3 map of $38\times38\times512$ size is transformed to a $38\times38\times4\times(c+4)$ output size.
2. The Conv7 map of $19\times19\times1024$ size is transformed to a $19\times19\times6\times(c+4)$ output size.
3. The Conv8_2 map of $10\times10\times512$ size is transformed to a $10\times10\times6\times(c+4)$ output size.
4. The Conv9_2 map of $5\times5\times1024$ size is transformed to a $5\times5\times6\times(c+4)$ output size.
5. The Conv10_2 map of $3\times3\times256$ size is transformed to a $3\times3\times4\times(c+4)$ output size.
6. The Conv11_2 map of $1\times1\times1256$ size is transformed to a $1\times1\times4\times(c+4)$ output size.

2.5.2 Default Bounding Boxes

Each of the m features map learns to detect objects at a particular scale, defined as follows:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m-1}(k-1), \quad k \in [1, m]. \tag{1}$$

For each scale, the network predicts boxes of 5 different aspect ratios: $\{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$. Then, the width and the height of the box can be calculated by the following equations:

$$w_k^a = s_k\sqrt{a_r} \quad and \quad h_k^a = \frac{s_k}{\sqrt{a_r}}. \tag{2}$$

In addition, the authors added one extra scale $s_k^{'} = \sqrt{s_k s_{k+1}}$ for the aspect ratio of 1 and finally, the center of each box is computed as follows: $\left(\frac{i+0.5}{|f_k|}, \frac{j+0.5}{|f_k|}\right)$, where $|f_k|$ is the size of the k-th square feature map and $i, j \in [0, |f_k|)$.

2.5.3 Training Strategy

During training, the authors assigned multiple default boxes that vary over size and aspect ratio to a ground truth detection with Jaccard overlap higher than 0.5. With this definition, a single detection may rely on multiple feature maps and with backpropagation, the weights of the network are updated accordingly.

The loss function being optimized is the following:

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)), \tag{3}$$

where N is the number of matched default boxes.

The localization loss is a smooth L_1 loss over the l,g offsets of the predicted l and the ground truth box g. The definition of this loss is similar to Faster R-CNN[8] and is defined as follows:

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k smooth_{L1}\left(l_i^m - \hat{g}_j{}^m\right)$$

$$\hat{g}_j{}^{cx} = \frac{\left(g_j^{cx} - d_i^{cx}\right)}{d_i^w} \qquad \hat{g}_j{}^{cy} \frac{\left(g_j^{cy} - d_i^{cy}\right)}{d_i^h} \tag{4}$$

$$\hat{g}_j{}^{w} = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j{}^{h} = \log\left(\frac{g_j^h}{d_i^h}\right)$$

where (cx, cy), w and h are the center offsets, the width and the height of the default box d respectively. The $x_{ij}^p = \{1, 0\}$ is an indicator for the i-th default box to the j-th ground truth box of category p.

The confidence loss is a softmax over the c classes:

$$L_{conf}(x, c) = -\sum_{i \in Pos}^{N} x_{ij}^p \log\left(\hat{c}_i{}^p\right) - \sum_{i \in Neg} \log\left(\hat{c}_i{}^0\right), \tag{5}$$

$$where \qquad \hat{c}_i{}^p = \frac{\exp\left(c_i^p\right)}{\sum_p \exp\left(c_i^p\right)}.$$

Closing remarks:

1. The authors used hard negative mining[18] to form a balanced training set with negative-positive ratio of $3{:}1$(based on the confidence score).
2. SSD struggles to locate small objects, because the input image of $300{\times}300$ size is relatively small and the early feature maps do not retain much information. On the other hand, it is very accurate for handling bigger object.
3. By using more feature maps the performance of SSD is increasing.
4. SSD is extremely sensitive to the bounding box size.
5. A proposed data augmentation method increased the accuracy on small objects by acting like a ”zoom in” effect. They used random crop, horizontal flip and some photo-metric distortions and the mAP improved by 8.8%.

2.6 YOLO9000: Better, Faster, Stronger

In this work, Joseph Redmon et al.[11] proposed some critical modifications so as to cure the low recall and localization errors of YOLOv1[9] but at the same time maintaining the high classification accuracy. YOLOv2[11] runs at 67 fps with 76.8 mAP on VOC2007[14] dataset and at $544{\times}544$ resolution runs at 40 fps and surpasses the performance of Faster R-CNN(ResNet)[8] and SSD[10].

In addition, the authors introduced a novel real-time object detection system that detects over 9000 different classes. It is called YOLO9000 and it was trained jointly on COCO[22] detection and ImageNet[23] classification datasets by using a hierarchical view of object classification that permits us to combine these distinct datasets.

2.6.1 YOLOv2

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?							✓	✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Source: [11]

Figure 2.10: Improvements in YOLOv1

Novel modifications in YOLOv1:

1. To improve the network convergence and reduce over-fitting dropout layers replaced by batch normalization[24]. With this change, the performance increased by more than 2%.

2. YOLOv1[9] is pre-trained for the classification task at 224×224 resolution and then, is fine-tuned at 448×448 for the detection task. This procedure, though, affects the training pipeline by forcing the network to even learn the new input image size. On the contrary, YOLOv2 is trained at 448×448 resolution for both tasks and the mAP improved almost by 4%.

3. Localization errors were prevalent in the previous work and as a consequence, the regression part was revisited. The prediction of box coordinates was a difficult task to learn directly and the fc layers replaced by conv layers. By using anchor boxes and predicting offsets, the recall increased from 81% to 88%.

   The input image resized to 416×416 resolution so that the output feature map contains a central cell. This cell is quite important for detecting bigger objects that are located in the center of the image. Regarding the class prediction, YOLOv1 assigns one class per cell for all the B bounding boxes. In contrast, in YOLOv2 each individual box has its own class, but the objectness score is remained untouched. The sub-sampling ratio here is 32 and the final feature map has 13×13 size.

   The final observation made by the authors was the problematic definition of offsets in Faster R-CNN[8]. During training, there was high jittering in the loss function in the first iterations. Faster R-CNN predicts offsets (t_x, t_y) and the center of the box is calculated as follows:

$$x = (t_x * w_a) - x_a$$
$$y = (t_y * h_a) - y_a,$$

   where (x_a, y_a) and (w_a, h_a) are relative to anchor a. If $t_x = 1$, then the box is shifted to the right by the width of the anchor, since the definition hasn't any constraints. In this way, we can end up with a box at any point in the image regardless of the location in the feature map.

   YOLOv2 solves this issue with a sigmoid activation function and now the offsets are bounded: $(t_x, t_y) \in [0, 1]$, relative to each cell.

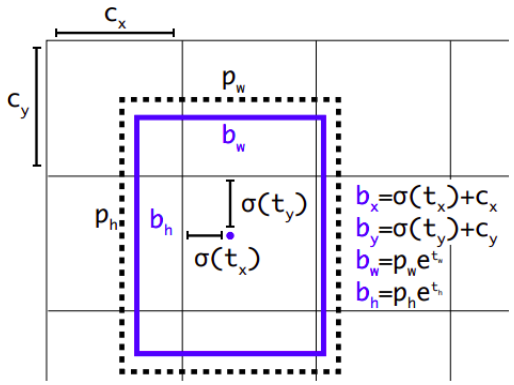
   The new prediction equations are the following:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \\ Pr(object) * IOU(b, object) &= \sigma(t_o), \end{aligned} \tag{1}$$

   where (c_x, c_y) is the offset from the top left corner of the image and (p_w, p_h) are the bounding box prior width and height, respectively(refer also to figure 2.11).

4. Had-picked anchors was a hasty selection. Instead, the authors run on training set k-means clustering to find better priors. They even found out that k-means with euclidean distance performs poorly, because larger boxes generate more error than the smaller ones. The main goal here is to pick priors with high IoU scores with respect to the true boxes and the ideal distance metric seem to be:

$$d(box, centroid) = 1 - IOU(box, centroid). \tag{2}$$



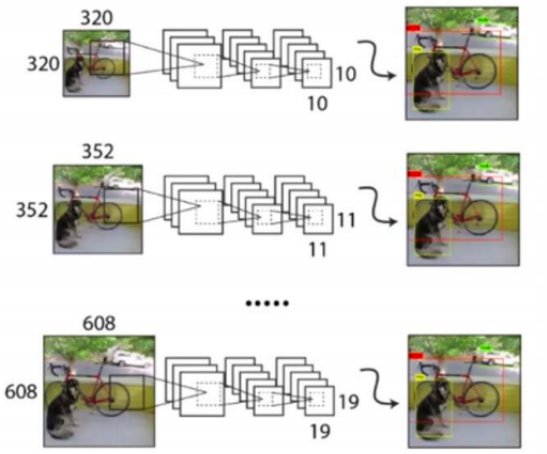
Source:[11]

Figure 2.11: The new anchors definition in YOLOv2

By computing the average IoU to the closest prior, the avg IoU of 5 centroids is 61.0 whereas 9 hand-picked anchors have 60.9. With 9 centroids the avg IoU is even higher at 67.2. YOLOv2 improved by 5% relative to YOLOv1 by employing anchors and k-means clustering.

5. To improve the localization performance in smaller objects, a pass-through layer was added that brings features from an earlier layer of $26\times26\times512$ resolution. This map is then transformed into a $13\times13\times2048$ feature map by stacking adjacent features into different channels and finally, it can even concatenated with the original 13×13 map. With this trick, the performance increased by 1%.

6. It is known that a convolution neural network can handle images of various scales while maintaining the same weights(filters and kernels remain unmodified and only the size of the feature maps is changed). YOLOv2 exploits this property and every 10th iteration the input image is resized to a specific range: $\{320, 352, ..., 608\}$. This range has been selected carefully, since every input image is down-sampled by a factor of 32. Such a modification can be interpreted as an augmentation technique. For example, an image of 352×352 resolution is transformed to a 11×11 feature map and the final prediction is a $11\times11 * B * (5 + C)$ tensor as captured in the figure below.



Source: Daniel Gordon's YouTube channel

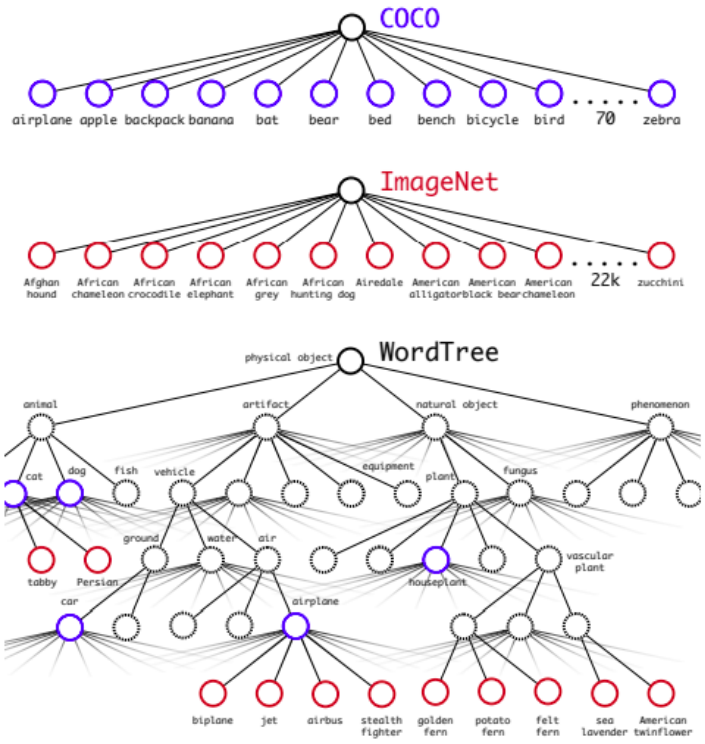
Figure 2.12: Multi-scale training

7. To reduce the test time, the authors, proposed a backbone called Darknet-19 that achieves 91.2% top-5 accuracy on ImageNet while VGG-16 achieves only 90%. It contains batch normalization, pooling, 3×3 and 1×1 layers and the number of channels is doubled after every pooling layer. In the final layers, the backbone contains one 1×1 layer with $7\times7\times1000$ output size and one global average pooling layer(performed per channel) with 1000 outputs for the classification task. For the detection task, the global avg pooling layer is replaced by a 1×1 layer.

2.6.2 YOLO9000

The authors even built a novel object detection system named YOLO9000 that is trained on COCO detection and ImageNet classification dataset, respectively. The network initially learns to detect common images and then, its vocabulary of categories is enriched by the classification set.

Achieving such a task might seem trivial, but one must examine meticulously the structure of these datasets. First of all, most of the detection datasets contain mainly generic labels, like ”bird” or ”dog”. On the contrary, ImageNet contains even breeds of dogs. However, common classification deep learning techniques assume classes are mutually exclusive and use a final softmax layer. Therefore, training a network with such a layer on both COCO and ImageNet seemed problematic.



Source: [11]

Figure 2.13: Combine datasets into a WordTree

The proposed solution is to build a hierarchical tree and sort these classes. The so-called WordTree(refer to figure 2.13) inspired by the WordNet[25], a complex language database that structures concepts and how they are related with a directed graph.

These structures are summarized in the following examples:

1. A ”cockatoo parrot” and ”macaw parrot” are both hyponyms of parrot which are both birds.
2. A cat is a mammal and an animal which are both synsets.

To build a WordTree, first we investigate a visual noun and look their paths to the root node ”physical object”. If this noun has a unique path, then the whole path is being added to the tree. For the nouns that have more than one path, the smallest one is being added. As a result, the classification task now predicts conditional probabilities in each node of the tree for the probability of each hyponym of that synset given that synset(see figure 2.14). For example, YOLO9000 predicts for the “terrier” node:

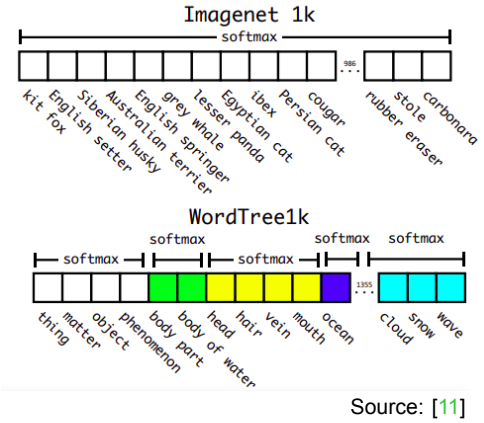
$$Pr(Norfolk\ \ terrier|terrier)$$
$$Pr(Yorkshire\ \ terrier|terrier)$$
$$...$$

One can even compute the probability of a node by following its path and multiplying the conditional probabilities. As an example, the authors mention the Norfolk terrier:

$$\begin{aligned} Pr(Norfolk\ \ terrier) = Pr(&Norfolk\ \ terrier|terrier) \\ &*Pr(terrier|hunting\ \ dog) \\ &*...* \\ &*Pr(animal|physical\ \ object), \end{aligned}$$

where $Pr(physical\ \ object) = 1$. During the training process, the network predicts values for all the intermediate nodes and if an object is classified as dog it should also be classified as an animal. For the detection task, we set $Pr(physical\ \ object) = objectness\ \ score$ and we traverse the tree down by picking the nodes with the highest probabilities to assign a class to each box.

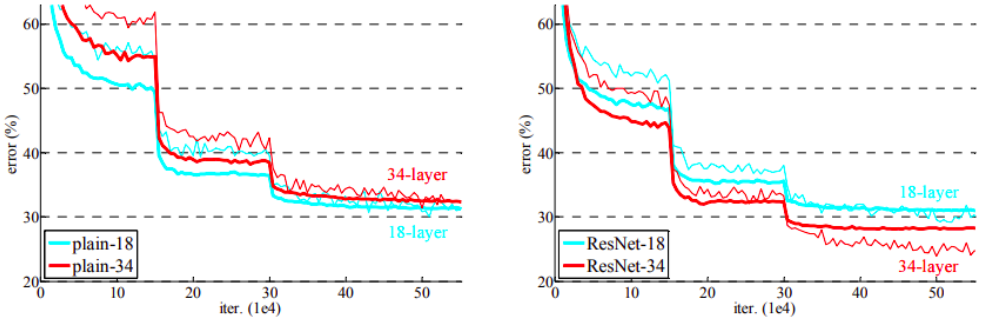
To train the whole network jointly, the train set is now a combination of classification and detection labels. For an object detection example, we backpropagate the loss without any modification. For a classification example, we backpropagate at or above the node of the label. For instance, if a label is a ”bird” we do not assign an error to further nodes, like ”canari”, as this bird might be a ”parrot”.



Source: [11]

Figure 2.14: ImageNet vs WordTree predictions

2.7 ResNet: Deep Residual Learning for Image Recognition



Source: [12]

Figure 2.15: Training(thin curves) and validation error(bold curves) in plain network and ResNet

At that time, the Deeper Neural Networks were winning various image recognition competitions and the Academia tried to answer if by stacking more and more layers on models the performance is always increasing. Unfortunately, they found out that there is a sweet spot and after a while the networks face a lot problems, such as vanishing or exploding gradients[26]. A key observation was that, the main source of this problem wasn't related to any kind of over-fitting. As we can see in the left image in figure 2.15, the training error is higher in the deeper network than in the smaller one.

To solve these issues, Kaiming He et al.[12] built a learning framework called ResNet to ease the training of Deep Networks and even win all the competitions on ILSVRC 2015 and COCO 2015 datasets. The authors approach this challenge as an optimization problem and inspired mainly by a simpler task: in theory, if we add to a deep neural network some extra layers that perform the identity mapping, then the network should have no higher error relative to the shallower one. But in practice, with the preexisting solvers, such a function is difficult to learn directly.

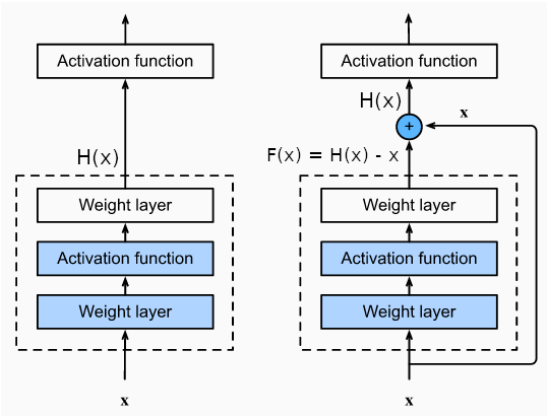


Figure 2.16: Plain and Residual blocks

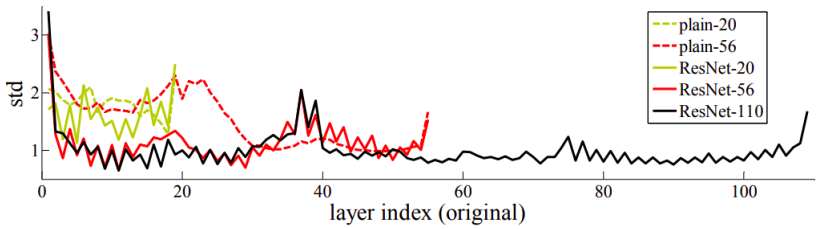
In ResNet, the typical building blocks(left image in figure 2.16) of neurons with weights and activation functions are defined slightly differently. Instead of learning a desired mapping H(x) of the input x, we learn a new F(x) = H(x) - x mapping and the original one is defined as: H(x) = F(x) + x(right image in figure 2.16). The new mapping H(x) is formed with the proposed "shortcut connections" that skip one or more layers.

More formally, if we use a ReLU activation function g(x), then the output a^{l+2} of the so-called residual block is computed as follows: $a^{l+2} = g(z^{l+2}+a^{l}) = g((W^{l+2}*a^{l+1}+b^{l+2})+a^{l})$.

The most significant property here is that, there is no need to learn extra weights and if we even desire the network to learn the identity mapping(H(x) = x), it is easier to ”push” the weights towards to zero($a^{l+2} = g(a^l) = a^l$) rather than learning this function directly. The intuition behind this formulation is that we hope the additional ResNet blocks do not harm the learned representation but even help a little bit.

In the Residual blocks, the F(x) and x vectors should have the same dimension, but this is not always the case. For that reason, the authors multiply the input a^l with an extra matrix: $W_s * a^l$. This matrix can have fixed values that add zero-padding to the input or even, it can contain trainable weights.

The ResNet framework enables us to train bigger networks more effectively and exploit the additional layers. The right image in figure 2.15 validates the previous statement and if we observe carefully, we can see that the 18-layer ResNet converges faster than the 18-layer plain network.



Source: [12]

Figure 2.17: Standard deviations of layer responses on CIFAR10[27]

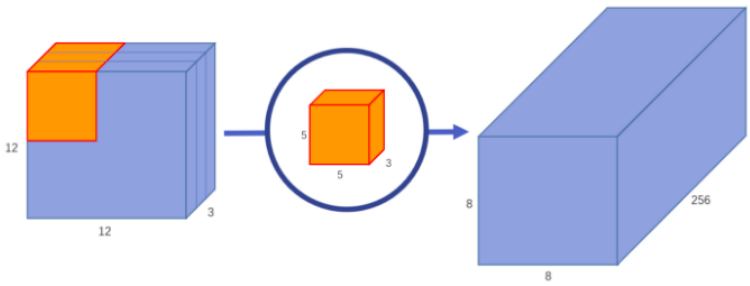
The authors even found out that the output’s responses per layer are smaller, as capture in figure 2.17, as the residual functions can be closer to zero.

At last, they managed to train a 1000-layer ResNet and minimize the training error, even though the performance is certainly worse than a 110-layer ResNet due to over-fitting.

2.8 MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Andrew G. Howard et al.[13] proposed a class of efficient models called MobileNets. In this work, they achieved to create models that are efficient with respect to size and speed. The typical convolutional layers replaced by depthwise separable convolutions and even two global hyperparameters were introduced in order to shrink uniformly the network's size. The depthwise seperable convolution is a form of factorized convolutions in which the conventional convolution is splitted into a depthwise and $1{\times}1$ pointwise convolutions.

To begin with, in a typical convolution operation(s = 1 and p = 1), N filters of size $D_K{\times}D_K{\times}M$ convolute the input $D_F{\times}D_F{\times}M$ feature map and produce an output of $D_F{\times}D_F{\times}N$ size with $N{\times}D_K{\times}D_K{\times}M{\times}D_F{\times}D_F$ multiplications. As an extra example you can refer to figure 2.18 and to be more precise, 256 filters(s = 1 and p = 0) of $5{\times}5{\times}3$ size generate an $8{\times}8{\times}256$ output map with $256 * (5 * 5 * 3 * 8 * 8) = 1.228.800$ multiplications.



Source: Chi-Feng Wang in Towards Data Science

Figure 2.18: Typical convolution operation

In MobileNets, we can prune the $N(= 256)$ term from cost by using depthwise separable convolutions that "separate the filtering from the combination step".

At first, depthwise convolution applies a single filter(s = 1 and p = 1) per each input channel. The input $D_F * D_F * M$ feature map is transformed into an output $D_F * D_F * M$ feature map with N filters of $D_K{\times}D_K{\times}1$ size and the total cost is $D_K * D_K * M * D_F * D_F$ multiplications. Following the previous example: 3 filters of $5{\times}5{\times}1$ size generate an $8{\times}8{\times}3$ output map with $5 * 5 * 3 * 8 * 8 = 4.800$ multiplications(top image in figure 2.19).

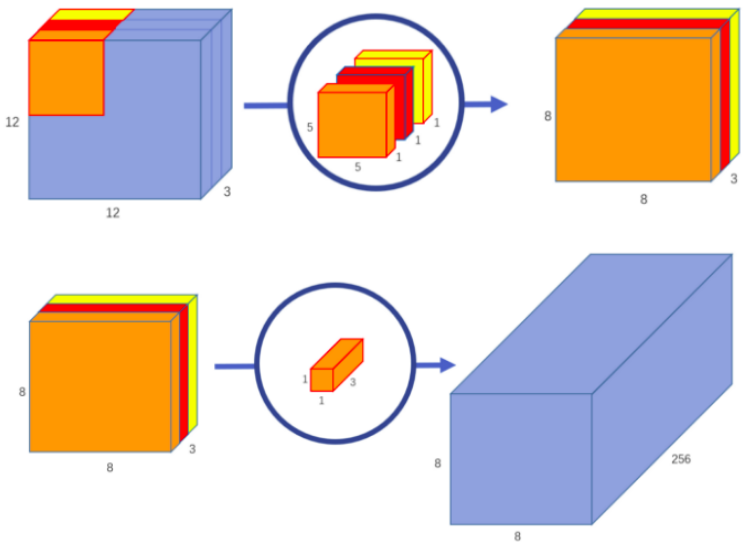
Finally, to combine these intermediate maps we use N filters of $1{\times}1{\times}M$ size to compute the final output $D_F{\times}D_F{\times}N$ feature map with $N * D_F * D_F * M$ multiplication cost. In the second picture in figure 2.19, the $8{\times}8{\times}3$ map is now elongated to an $8{\times}8{\times}256$ output map with $256 * (8 * 8 * 3) = 49.152$ multiplications.

As a result, the input feature map is transformed only once by depthwise convolutions rather than N(=256) times as in the original convolution operation. The cost from $1.228.800$ multiplications is reduced to $53.952 = 49.152{+}4.800$ multiplications. More formally, by using the original definition of the paper, depthwise separable convolutions costs:

$$D_K * D_K * M * D_F * D_F + N * D_F * D_F * M \tag{1}$$

and we get a total reduction in computation of:

$$\frac{D_K * D_K * M * D_F * D_F + N * D_F * D_F * M}{N * D_K * D_K * M * D_F * D_F} = \frac{1}{N} + \frac{1}{D_K^2}. \tag{2}$$



Source: Chi-Feng Wang in Towards Data Science

Figure 2.19: Depthwise separable convolutions

The original MobileNets(28 layers) use 3×3 filters for the depth wise convolutions and after each pointwise and depthwise convolutions, the authors added batchnorm[24] and ReLU layers. More precisely, only the first layer is a conventional convolution and the remaining are depth wise separable. At last, there is one average pooling and one fc layer which feeds into a softmax layer for classification. The authors also suggest during training less regularization and very little or no weight decay, since the network is small and less prone to over-fitting.

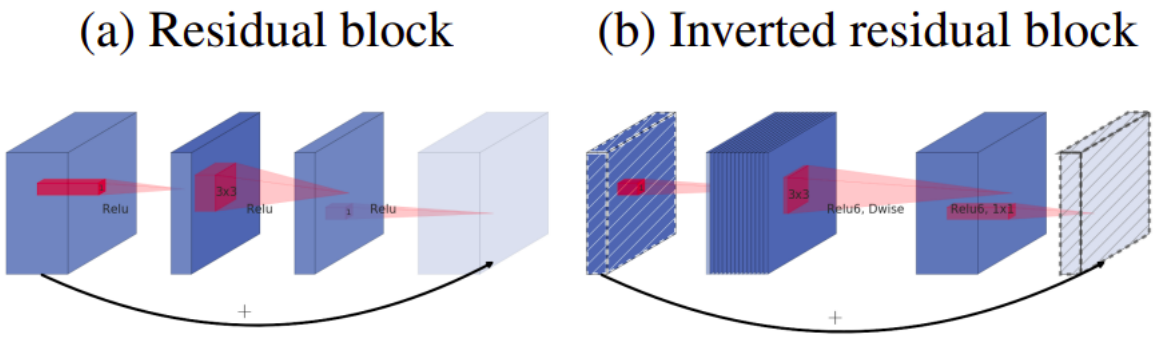
MobileNets[13] not only decreased the number of the parameters, but two extra hyper-parameters enable us to make the network even smaller. The first hyperparameter is the width multiplier α that thins the network. In particular, each M input and N output channels are truncated to αM input and αN output channels, respectively and the cost is reduced quadratically by α^2. The last hyperparameter is the resolution multiplier ρ which regulates the input resolution. With these hyperparameters the overall cost is now:

$$D_K * D_K * \alpha M * \rho D_F * \rho D_F + \alpha N * \rho D_F * \rho D_F * M. \tag{3}$$

Concluding remarks:

1. The original network with 4.2 million parameters is only worse by 1% on ImageNet dataset than the fully convolutional network with 29.3 million parameters.
2. MobileNets have comparable accuracy with VGG-16[19], but at the same time, they are 32 times smaller and 27 times less computational demanding.
3. SSD(MobileNets)[10] outperforms Faster R-CNN(MobileNets)[8] on COCO2016 by 3%.
4. By removing 5 point wise separable layers the mAP on ImageNet set is worse by 3% relative to MobileNets with width multiplier of $\alpha = 0.74$.

2.9 MobileNetV2: Inverted Residuals and Linear Bottlenecks



Source: [1]

Figure 2.20: The difference between residual and inverted residual blocks

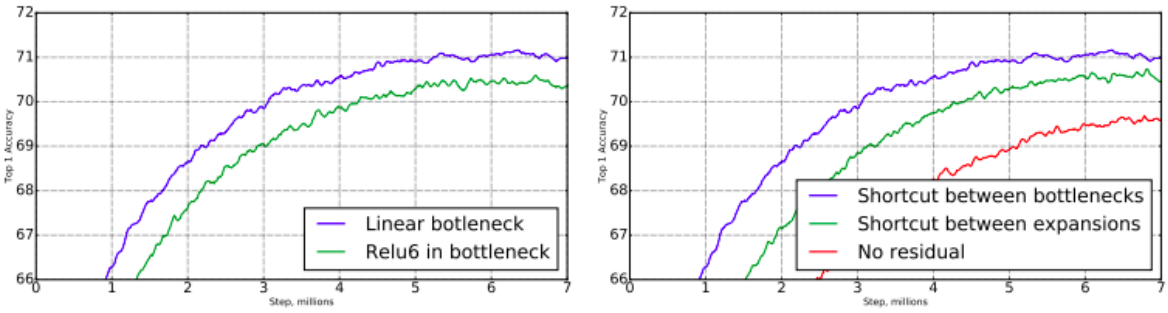
Mark Sandler et al.[1] improved the prior work of MobileNets[13] by introducing novel inverted residuals and linear bottlenecks. The network's parameters reduced by more than 75% and the mAP on ImageNet increased by 1.6%, relative to MobileNets[13].

MobileNetV2[1] is mainly inspired by the following properties of the ReLU activation function:

1. If the ReLU(x) transformation has non-zero output volume, then that ReLU is the identity transformation of x: $x = ReLU(x)$.
2. If ReLU(Bx) with $B \in \mathbb{R}^{m \times n}$ has a non-zero volume, then it performs to the input x a linear transformation.
3. If ReLU(x) collapses a depth channel, then the information might be lost.
4. If the input $x \in \mathbb{R}^n$ can be embedded into a lower dimension$(n-1)$, then the first property is always true, as the starting volume is 0.
5. The equation $y_0 = ReLU(Bx)$ has a unique solution with respect to x if and only if y_0 has at least n non-zero values and there are n linearly independent rows of B that correspond to non-zero coordinates of y_0. More intuitively, if we have sufficient channels$(m \gg n)$, then the ReLU is invertible and the information might be preserved.

Motivated by these properties and even from the residual blocks[12], the authors created the so-called inverted residuals with linear bottlenecks(right image in figure 2.20). Instead of compressing the input feature map and connect the layers with high number of channels, they expanded the input map and used shortcuts directly between the bottlenecks, since bottlenecks contain all the information. Additionally, they used linear activation functions to the input bottleneck to prevent non-linearities from erasing much information. In this way, the first 1×1 layer acts as a decompressor that first restores the data to its original form(as possible) before using filters with ReLU functions(property 5). Then, the typical convolution is replaced by a depthwise layer to perform the filtering. Finally, the last 1×1 layer "makes" the data compact again. On the other side, the classical residual blocks filter a low-dimensional tensor with ReLU and as a result we lose lot of information.

To be more precise, the input bottleneck of size $h\times w\times k$ is transformed with 1×1 filters of linear activations to an $h\times w\times(tk)$ output map. Next, the depthwise convolutions filter the previous map and predict a $\frac{h}{s}\times\frac{w}{s}\times(tk)$ map. Lastly, a 1×1 layer of ReLU6 generates a final $\frac{h}{s}\times\frac{w}{s}\times(k')$ map.



Source: [1]

Figure 2.21: The impact of non-linearities and linear residuals in MobileNetsV2

The authors validated that the proposed ideas enhanced the performance(see figure 2.21) and at last, built an improved version of SSD[10] named SSDLite. They replaced the prediction layers with separable convolutions and added dropout and batch normalization layers. The network parameters reduced from 14.8 million to 2.1 million.

3. LEADER-FOLLOWER SCHEME

In this chapter, at first, we define formally our Leader-Follower Scheme based on [28], but without input constraints and then, we discuss its main components and the software design.

In the next chapter, we continue by presenting our approach to this Scheme.

3.1 Problem Statement

Definition 1 Set d > 0 and let $R_0 = P_0^T, R_1 = P_1^T$ be two robots. We say that R_0 and R_1 are in d-formation with leader R_0 at time t, if

$$P_0(t) = P_1(t) + d \tag{1}$$

and, simply, that R_0 and R_1 are in d-formation with leader R_0, if (1) holds for all $t \geq 0$. Moreover we say that R_0 and R_1 are asymptotically in d-formation with leader R_0 if

$$\lim_{t\to\infty} P_0(t) - (P_1(t) + d) = 0. \tag{2}$$

Definition 1 states that two robots R_0, R_1 are in d-formation with leader R_0 if the position P_1 of the follower R_1 is always at distance d from the position P_0 of the leader R_0, that is the position P_0 of the leader remains fixed with respect to the follower reference frame.

An equivalent way to state the Definition 1 is the following:
set the error vector

$$E(t) = P_0(t) - (P_1(t) + d)\,, \tag{3}$$

then R_0 and R_1 are in d-formation(asymptotically) if and only if $E(t) = 0$, $\forall t \geq 0$(or, $\lim_{t\to\infty} E(t) = 0$).

Problem 1 Set $d > 0$ and let R_0, R_1 be two robots. Find necessary v_1 and ω_1 controls for R_1 such that R_0 and R_1 are in d-formation(asymptotic) with leader R_0.

For our Leader-Follower Scheme we set $d = 1.7m$, so as to prevent any contact between the robots during the real experiments.

Problem 2 The percentage of false detections in 1.7-formation must be lower than 5%.

3.2 System Overview

Our Leader-Follower Scheme consists of multiple and complex components. First of all, there are two mobile robots: the Summit-XL(follower) and the Pioneer 3-AT(leader). The Pioneer 3-AT steers with a joystick(operated by humans) and the Summit-XL follows the Leader by using a position controller. Additionally, an extra joystick is connected to Summit-XL for safety reasons(but no action was taken with the additional joystick during the experiments).

An essential device of our scheme is also, one RGB-D Camera which is mounted on Summit-XL for perception purposes. To be precise, we process the RGB information with MobileNets[1] algorithm and detect the Leader in the image frame. Then, we compute the

3D relative position (x, y, z) of the Pioneer 3-AT by using the depth information. At last, we plug these coordinates to a position controller to calculate the appropriate Follower commands (u, ω).

Furthermore, it is a common practice to exploit a GPU card while processing RGB-D data. For that reason, all the heavy computations in the RGB-D frames as well as the controller loop take place in an extra desktop computer. Though, to transmit the final (u, ω) controls from the desktop to the Follower's computer, we connect both devices with an Ethernet cable and the data exchange is accomplished with the help of a ROS package called multi-master_fkie(more in the next paragraph). Finally, we monitor all actions with an additional laptop which is connected to the same network(provided by Summit-XL) via Wi-Fi.

Last but not least, these robots need a "brain" to operate and follow our instructions. A meta-operating system named Robot Operating System(ROS) is installed on their computer which provides libraries, tools, device drivers, message-passing and more. In the ROS eco-system, the folders that contain our source code are called packages and each running process is called a node. The nodes can communicate with each other via topics by sending messages. Topics are named buses(such as sockets or pipes) and implement an asynchronous communication pattern as a message is posted from one node(publisher) to the topic and any nodes(subscribers) which listen to that particular topic can receive messages. More information about ROS can be found at: `http://wiki.ros.org/ROS/Introduction`.

In the following subsections, we discuss the hardware specifications of the Leader-Follower Scheme as well as the software design.

3.2.1 Leader: Summit-XL

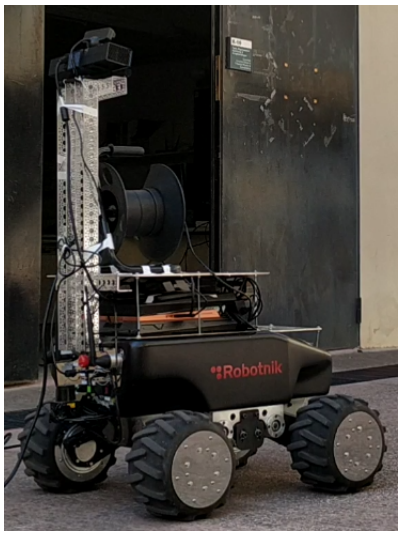


Figure 3.1: Summit-XL mobile robot

The Summit-XL robot was built from Robotnik. It has skid-steering kinematics based on 4 high power motor wheels. It weights 45 Kg and can carry up to 20 Kg of payload. The maximum speed is 3 m/s with 5 hours autonomy and needs only 2 hours of charging. Additionally, it contains an IMU with 3d accelerometer, gyroscope, magnetic field and even a GPS.

3.2.2 Follower: Pioneer 3-AT

Figure 3.2: Pioneer 3-AT mobile robot

The Pioneer 3-AT robot was built from Adept Robots. It is a small four-wheel, four-motor skid-steer robot. It weights 12 Kg and can carry up to 7 Kg of payload. The maximum speed is 0.7 m/s and with 3 batteries can operate for 4 hours. The charging time is about 12 hours.

3.2.3 RGB-D Camera: Kinect Sensor v2

Figure 3.3: Kinect sensor V2

The Kinect Microsoft camera is a high quality, low-cost RGB-D sensor that includes one RGB camera, one IR camera and one IR emitter. The depth estimation is based on the Time of Flight(ToF) Measurement Principle and its technical specifications are:

- Color Camera = 1920×1080 @30FPS
- Depth(IR) Camera = 512×424 @30FPS
- Field of View of depth image = 70 Horizontal and 60 Vertical
- Field of View of color image = 84.1 Horizontal and 53.8 Vertical
- Range = 0.5m - 4.5m

3.2.4 Laptop: HP 15ac110nv

Figure 3.4: Laptop

The main laptop to monitor the two robots and the desktop computer has an i7 6500U CPU @1.50 GHz(4 cores and 2 threads per core) and a 6GB RAM @1600MHz.

3.2.5 Desktop: Asus Rog GR8 II-T005Z

Figure 3.5: Desktop computer

The desktop computer that processes the RGB-D frames and estimates the Follower's velocities has an i7 7700 CPU @3.60 GHz(8 cores and 2 threads per core) and an 8GB RAM @2400MHz. It is also accompanied with a GeForce GTX1060 3GB GPU card.

3.2.6 Software Design

Software Requirements:

- ROS distribution: melodic.
- Tools and libraries for a ROS Interface to the Kinect: iai_kinect2.
- Object Detection: tensorflow 1.15, CUDA 9.0 and TensorFlow Official Models 1.13.0.
- The required python packages can be found inside the BSc_thesis/code folder at requirements2.txt and requirements3.txt files.
- ROS package to establish and manage the communication between the Desktop and the Follower's computer: multimaster_fkie.
- Robotic Simulation to tune the controller: summit_xl_sim.
- Labelling tool: labelImg.

∗The code has been tested on Ubuntu 18.04.5 LTS in Python 2.7.17 and Python 3.6.9.

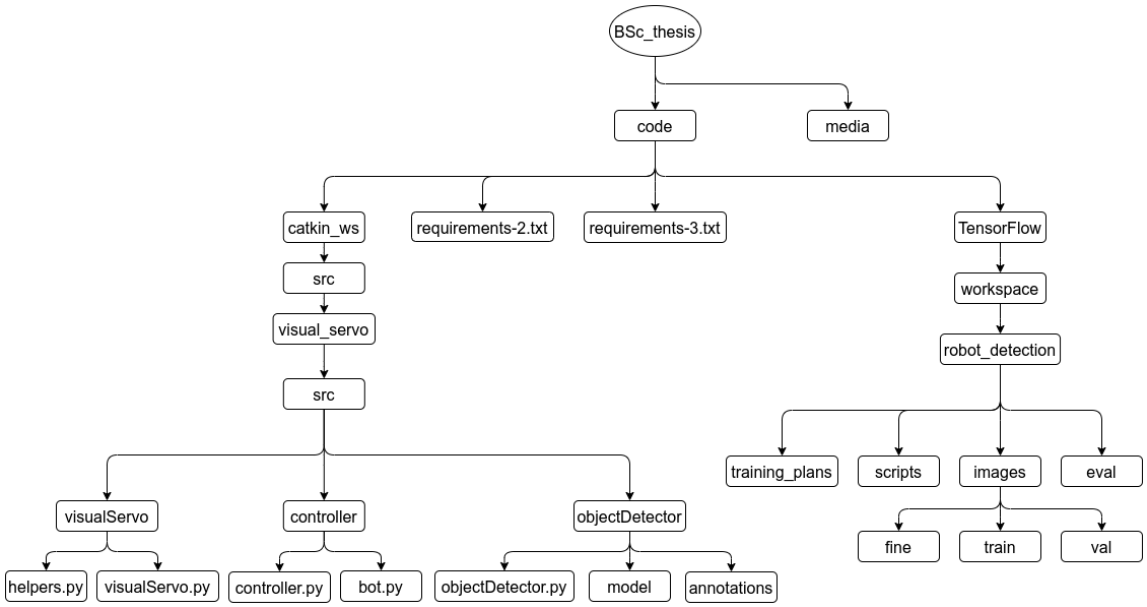


Figure 3.6: Folders Structure

The Folders Structure of our Leader-Follower Scheme is captured in figure 3.6 and to be more precise:

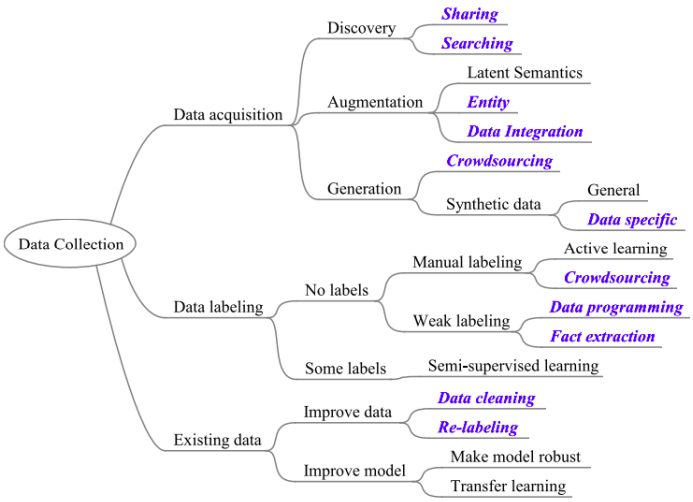
- In the media folder, we deliver videos and pictures from the real experiments.
- The images folder contains real and labeled Pioneer 3-AT examples.
- The visualServo.py file is the main module of the Scheme.
- The objectDetector folder contains a detection python module, a ”frozen” tensorflow model and the Leader’s annotations.
- The controller folder contains a controller.py module and a bot.py module that steers during the robotic simulation a Leader robot.
- The eval folder contains the evaluation results on the validation dataset for each training job.
- The training_plans folder contains configuration and checkpoint files.
- In the scripts folder you can find pre-processing and image augmentation python modules.

4. METHODOLOGY

4.1 Leader Detection

Now that we hold a high-level of understanding on the Leader-Follower Mobile Robot Scheme, we continue by breaking down the problem into smaller parts. In this section, we attempt to solve the Leader detection problem. The end goal here is to process the RGB frames and predict bounding box coordinates in the image plane. The go-to method to achieve state-of-the-art performance is to train a Deep Neural Network. The Deep Learning Life Cycle, though, consists of numerous stages that are revisited constantly, but we present our work in a structured manner without going ”back and forth”. We begin with the data collection process, then with the training pipeline and finally, we validate the detection system in multiple experiments with mobile robots.

4.1.1 Data Collection



Source: [29]

Figure 4.1: Data Collection overview

Data Collection is a vital component of the Deep Learning Life Cycle and as stated in [29], it consists of data acquisition, data labeling and improvement of models or existing data(refer also to figure 4.1).

To begin with, the main challenge in our task is to find labeled data. To our knowledge, there aren't any available open-source datasets for the Pioneer 3-AT and only animation pictures or images mainly on white background can be scrapped from the web. In addition, we plan to use the detection system in moving robots. Therefore, we gather real data with the Microsoft Kinect RGB-D Camera to ensure a high detection accuracy and we even open-source the dataset for potential research.

Initially, we recorded RGB frames of 960×540 resolution by using the iai_kinect2 ROS package at 3fps(with a laptop) while moving both robots with the joysticks. The majority of these pictures were taken at the location in which the Leader-Follower Scheme will operate during the final experiments. This area is a shady alley surrounded by walls and doors, as captured in pictures a-b and d-e in figure 4.2. Then, we enriched the dataset

with indoor pictures from our lab(refer to picture c in figure 4.2). This step is quite crucial if we want to avoid false positive predictions, since the alley is exploited by other labs and it is quite common to find new objects/devices.

Moreover, during training and testing we found out that, the detector struggled with shady or bright examples. Thus, we gathered additional data and saved them in the /images/fine folder.

The final dataset is divided into three folders: /images/train with 1722, images/fine with 93 and images/val with 919 labeled images, respectively. One that aims to use our set should also take into consideration that, the Pioneer 3-AT isn't crossing the image boundaries.

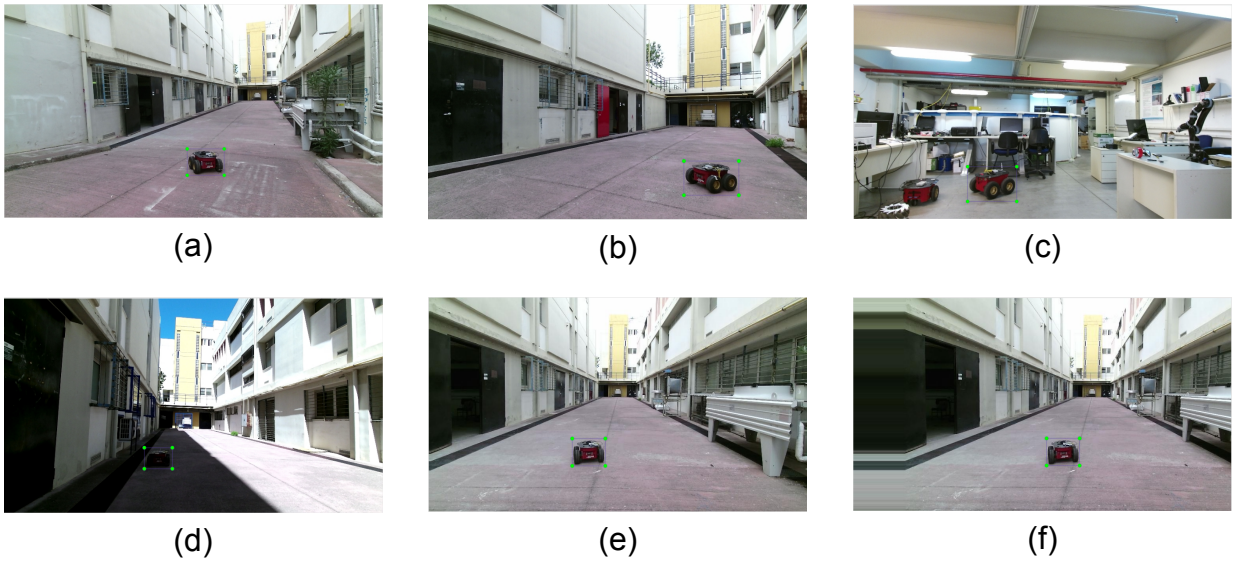


Figure 4.2: Dataset overview

As it can be perceived, our dataset is quite small to train from scratch a Deep Neural Network, as they typically require large amounts of training data due to the number of the parameters.

To overcome this challenge, we adopted the following techniques:

1. Transfer learning: we used the pre-trained neural network ssd_mobilenet_v2_coco from the Tensorflow Models API. Even though COCO hasn't any robot classes, we can benefit from the first layers and save a significant amount of time from training the network with random weights, as they can detect edges efficiently.

2. Data Augmentation: to increase the number of our training examples we augmented the dataset with the help of the python imgaug package. In scripts/aug.py, an affine transformation translates images/train examples in x-axis by 20%(see pictures e and f in figure 4.2). In the same spirit, scripts/aug1.py translates images/fine examples in x-axis by 20% and in y-axis by 2%. We picked these ranges carefully so as to avoid "infecting" a lot the relative perspective between the Leader and the Follower. Finally, with these transformations, we ended up with 56242 training examples.

Last but not least, we deliver some extra pre-processing scripts to handle efficiently all of these data:

- Firstly, the changePathXml.py and renameXml.py modules can rename, change the location of the training examples or even edit the fields of the annotation(.xml) files.

- Secondly, during the augmentation process it is quite common to end up with cross-boundary boxes. To remove the problematic cases, at first we can run the module validXml.py to generate a .txt file with the names of the invalid examples and then, with the deleteFiles.py module we can read the .txt and delete these examples.
- Finally, any duplicates examples should be eliminated from the training set. Such an operation might seem trivial, but image similarity is still an open problem in computer vision. A common approach is to use an image hashing algorithm. We found out that pHash is very accurate and fast. The pHash algorithm is summarized in the following pseudo python code and for more information you can check the module imageSimilarityFast.py.

```
def imageSimilarity(image_1, image_2):

    image_1 = rgbToGray(image_1)
    image_2 = rgbToGray(image_2)
    image_1 = resize(image_1, 9, 9)
    image_2 = resize(image_2, 9, 9)

    # Scan rows and columns and produce two lists of length 8x8=64                #
    # Lists contain 0, 1 values                                                   #
    # col_hash list: 1 means the pixel intensity is increasing in the y direction #
    # row_hash list: 1 means the pixel intensity is increasing in the x direction #
    # This type of hashing identifies the relative gradient direction             #
    row_hash_1, col_hash_1 = ([] for i in range(2))
    row_hash_2, col_hash_2 = ([] for i in range(2))

    for y in range(8):
        for x in range(8)

            # Scan rows #
            if image_1[y * 8 + x] < image_1[(y * 8 + x) + 1]
                row_hash_1.append(1)
            else:
                row_hash_1.append(0)

            # Scan columns #
            if image_1[y * 8 + x] < image_1[(y * 8 + x) + 8]
                col_hash_1.append(1)
            else:
                col_hash_1.append(0)

            # Hashes for image_2 are omitted #

    # Concatenate the hashes #
    image_1_hash = row_hash_1.append(col_hash_1)
    image_2_hash = row_hash_2.append(col_hash_2)

    # Count the number of different bits between the two hashes #
    diff = len([(i) for i in range(64 * 2) if image_1_hash[i] != image_2_hash[i]])
    if diff < threshold:
        return True # Duplicate
    else:
        return False
```

To conclude this section, Data Collection is a significant component of the Deep Learning process and as we discuss in the next chapters, the mentioned approach achieves more than sufficient performance in fixed robotic environments. On the other hand, real life autonomy requires more sophisticated methodologies. As stated by the director of AI in Tesla Andrej Karpathy: ”we need large, varied and real datasets and the labelling process can’t be done manually”. That is why the have developed more than 50 labelling projects and they even can gather incredible amounts of data from the fleet. On the other side, a rapidly increasing area in computer vision is self-supervised learning, where the Academia seeks methods to train Neural Networks without the need of labeled data.

4.1.2 Training and Validation

The next step of the Deep Learning process is to train the ssd_mobilenet_v2_coco model and evaluate its performance on the validation dataset, before testing the detection system in real experiments with the Pioneer 3-AT and Summit-XL.

To begin, we trained ssd_mobilenet_v2_coco on Google Colaboratory and imported our dataset from the Google Drive(refer also to train_collab.ipynb inside the TensorFlow folder). The total time needed to train the whole network is about 2 to 3 days.

Next, inspired by YOLO9000[11], we performed k-means clustering to find good anchors. The objective here is to partition all the bounding boxes to the closest centroid by using the IoU distance metric: $d(box, centroid) = 1 - IOU(box, centroid)$. The implementation is summarized in the following pseudo python code and the complete source code can be found in scripts/kmeans.py module.

```
def findAnchors(boxes):

    # Shape: (num of boxes, 2) => width, height #
    numBoxes = boxes.shape[0]

    # Choose randomly the initial clusters #
    clusters = boxes[np.random.choice(numBoxes, k, replace=False)]

    # Map each box with one cluster #
    currClusters = np.zeros((numBoxes,))
    prevClusters = np.zeros((numBoxes,))

    while True:
        for box in range(numBoxes):
            # Assign the closest cluster to the current box #
            distances = 1 - iou(boxes[box], clusters)
            currClusters[box] = np.argmin(distances)

        # Terminate condition: nothing changed #
        if (prevClusters == currClusters).all():
            break

        # Update clusters #
        for cluster in range(k):
            clusters[cluster] = np.median(boxes[currClusters == cluster], axis=0)

        prevClusters = currClusters

    # Return the aspect ratios of the centroids #
    ratios =  clusters[:, 0] / clusters[:, 1]

    return ratios
```

We performed k-means clustering in both training(+fine) and validation sets. Additionally, we run k-means multiple times to produce a consistent output, as the algorithm is sensitive to the initial random choice of clusters. The final values of aspect ratios of the anchors are: $\{0.7433, 0.79, 0.8133, 0.8233, 0.8433\}$.

During training, we picked a batch size of 16(the learning process was a bit unstable with smaller batch sizes) and trained the model for 130K steps(or 36 epochs) with Adam[30] optimizer. The number of iterations might seem relative small but as we discuss later, the training set contains mainly augmented images that add a little information as opposed to having real data and the model begins to converge after 80k steps.

We initially started the training job with a small learning rate and increased it gradually. This schedule is commonly used in pretrained models and even, it has been observed that Neural Networks generalize better with small initial learning rates. We also noticed that SGD slightly outperforms the Adam optimizer, but we sticked with Adam to reduce the training time. For more details, the reader can refer to the following articles: The 1cycle policy, How Do You Find A Good Learning Rate, Comparison of Optimizers and A Recipe for Training Neural Networks.

With respect to the input size, we found out that the default 300×300 input resolution of the network generalizes(during the experiments) and performs by 2% worse on validation set than the 512×512 resolution for the original 960×540 RGB frame, as captured in figure 4.3.

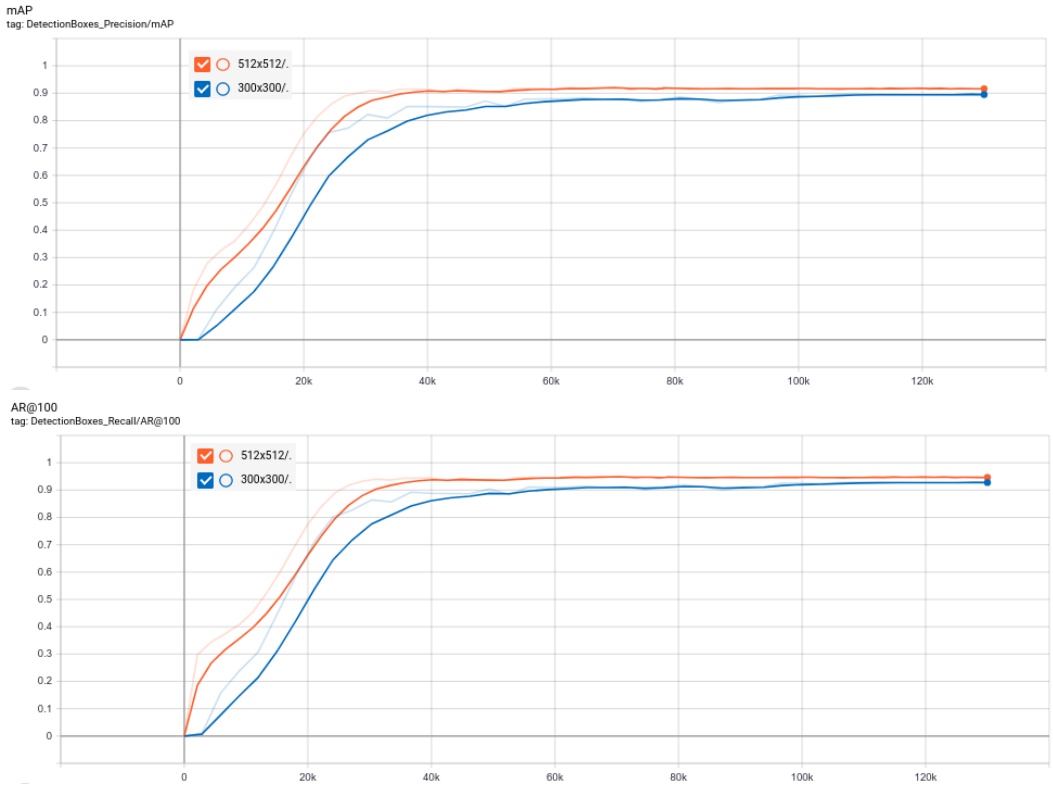


Figure 4.3: Performance at 512×512 and 300×300 input resolution

The configuration files of these plots can be found in training_plans/1 and training_plans/5 folders, respectively.

To gain the intuition behind the MobileNets[1], we tweaked most of the parameters in the configuration file and after a lot of experimentation, we present in figure 4.4 the most significant remarks of the detection model on the validation set. If you desire to reproduce these TensorBoard plots you can copy and paste in a terminal the first command from the Tensorflow/.../commands.txt file.

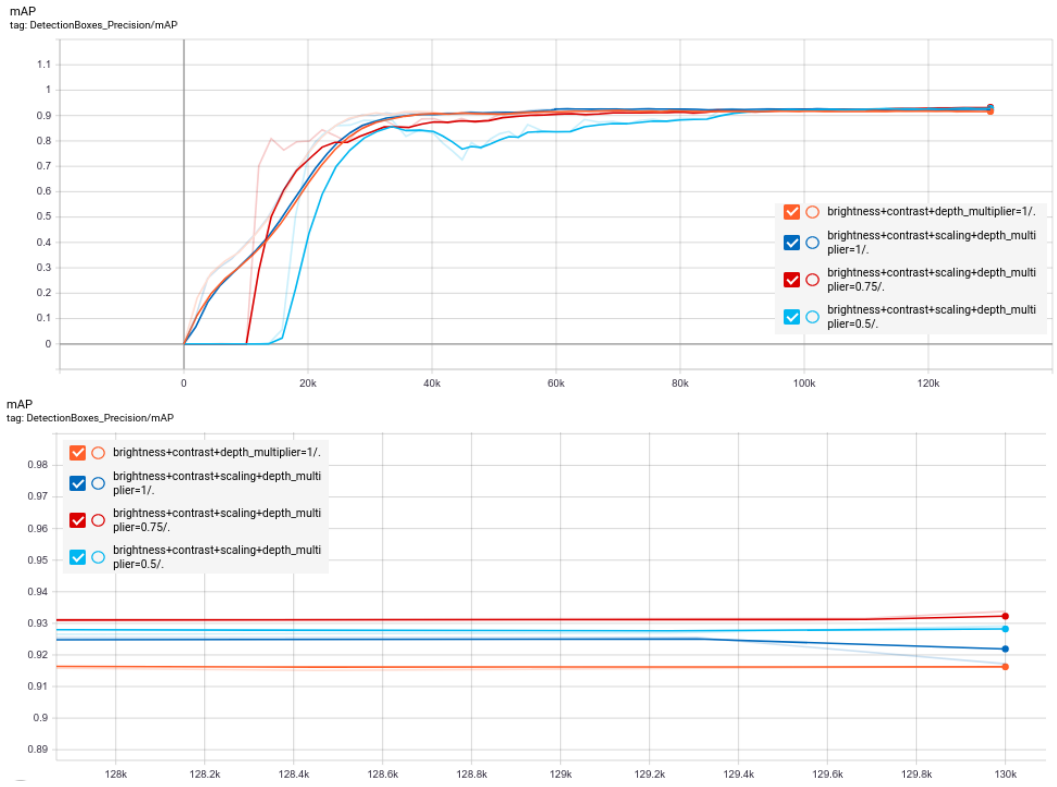


Figure 4.4: The impact of various training jobs on mAP

At first, to increase the robustness of the network in different lighting conditions we employed multiple pre-processing techniques provided by Tensorflow, such as random brightness by 35% and random contrast by 30%.

Then, as we can see in the figures above, transfer learning helped us to exploit the early layers by detecting edges efficiently. The mAP even at the very first iterations is increasing instantly and smoothly in the original pretrained models on COCO dataset with $depth\_multiplier = 1.0$. On the other hand, models 3 and 4 struggle at the beginning to learn with random weight initialization.

Additionally, we observed that random image scaling increased the mAP between model 1 and model 2 by 1%. As for the $depth\_multiplier$, model 3 with $\alpha = 0.75$ achieves the best performance on the validation set with 0.93% mAP and 0.96% AR@100. In general deeper networks outperform the smaller ones but in our scenario, the dataset contains only one class and the experimental environment isn't packed with objects. Nevertheless, there is always a sweet spot and the mAP degrades from model 4 with $\alpha = 0.5$ and afterwards.

Finally, we exported a frozen model from the best model and build a python module called catkin_ws/.../objectDetecor/objectDetector.py to detect the Pioneer 3-AT. Last but not least, inspired by the realtime_object_detection repository and the article Optimize NVIDIA GPU performance for efficient model inference by Qianlin Liang, we removed the post processing operations from the original graph and placed them in the CPU device to

increase the test time. The overall system(including the controller loop) runs at 19Hz on GeForce GTX1060 3GB GPU card.

4.1.3 Experiments

Before we continue with the Leader's position estimation, we tested the detector in real experiments on Pioneer 3-AT to secure its effectiveness. A significant portion of these experiments is captured in the table below and we even deliver supplementary videos that are located in the media/modelEvaluation/ folder.

Table 4.1: Experimental results about the detection system

Experiment	Total Frames	False Detections
media/modelEvaluation/1	629	00
media/modelEvaluation/2	2398	09
media/modelEvaluation/3	529	00
media/modelEvaluation/4	1301	00
media/modelEvaluation/5	1604	00
media/modelEvaluation/6	59	01
media/modelEvaluation/7	233	03

Experiments 1-5 are captured at 14 fps and took place at the location in which the Leader-Follower Scheme will operate during the final experiments. We also run some additional experiments, 6-7 at 3 fps, in unseen interior sections in our department to test the degree of generalization.

The results in table 4.1 look promising and we can, without hesitation, continue with the remaining components of the Scheme.

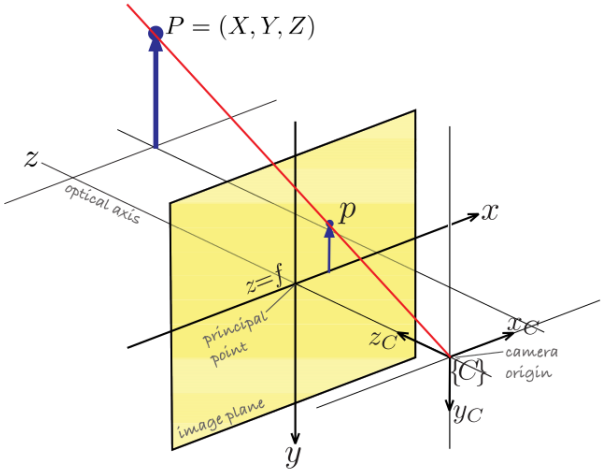
For more details about the experiments, such as the setup used, refer to chapter 5.

4.2 Position Estimation

So far, we have accomplished to detect the Pioneer 3-AT in the image plane by predicting pixel coordinates $(x_{min}, y_{min}, x_{max}, y_{max})$. However, the Leader-Follower Scheme requires the 3D coordinates (x, y) of the Leader with respect to the Summit-XL frame. Needless to say, images are a 2D representation of the world and we lose information corresponding to the third dimension. Though, to recover this information we can combine the Depth and the RGB frames of the Kinect.

In the following sections, we discuss more extensively the previous concept. At first, we present the fundamental principals of the Image Formation theory and then, we apply these principles to estimate the Leader's relative position.

4.2.1 Image Formation



Source: [31]

Figure 4.5: The central perspective imaging model

The central perspective imaging model(see figure 4.5) describes how 3D World points get projected into the image plane. With similar triangles, we can associate a 3D point $P = (X, Y, Z)$ to a 2D image point $p = (x, y)$:

$$x = f\frac{X}{Z} \quad and \quad y = f\frac{Y}{Z}, \tag{1}$$

where $z = f$ is the location of the image plane.

If we also convert the 2D point in the homogeneous form, the abode equations can be written in a compact matrix form:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

In addition, digital cameras capture the 3D world with a $W \times H$ grid of light sensitive elements, the photosites. Each of these photosites corresponds to a pixel or a tuple (u, v) of integers. By convention the origin is at the top-left corner(see figure 4.6) of the image plane and we can transform a 2D image point $p = (x, y)$ to a pixel point with the following equations:

$$u = x + o_x \quad and \quad v = y + o_y. \tag{2}$$

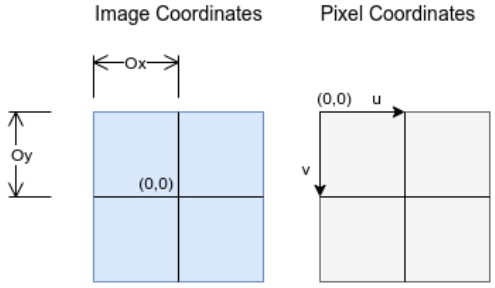


Figure 4.6: Image and pixel coordinate systems

However, most of the cameras have different scale factors in x and y axis and we do not necessarily have square pixels. Under these circumstances, the above transformation is reformed to:

$$u = \frac{1}{s_x}x + o_x \quad and \quad v = \frac{1}{s_y}y + o_y. \tag{3}$$

At last, we can even convert 3D coordinates directly to pixel coordinates by combining the equations (2) and (3) as follows:

$$u = \frac{1}{s_x}f\frac{X}{Z} + o_x \quad and \quad v = \frac{1}{s_y}f\frac{Y}{Z} + o_y. \tag{4}$$

The (s_x, s_y, o_x, o_y, f) parameters in the equation above are innate characteristics of the camera and can be obtained with a Camera Calibration process. These parameters also form the so-called Camera Matrix:

$$K = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix},$$

where $f_x = f * s_x$ and $f_y = f * s_y$.

Finally yet importantly, transformations such as (4) can't be retrieved in conventional RGB cameras, as the depth information is missing. But fortunately for us, as we describe in the next section, Kinect provides depth measurements and we can overcome this issue.

4.2.2 Leader's Position

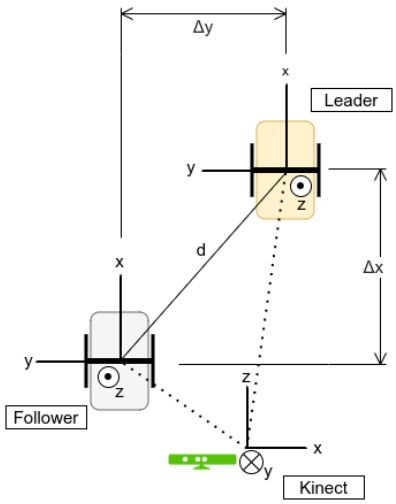


Figure 4.7: Leader-Follower coordinate systems

In this section, we apply the Image Formation theory to estimate the Leader's relative 3D position $(\Delta x, \Delta y)$ with respect to the Summit-XL(see figure 4.7). As for the Leader's position, we define to be the 3D coordinates of the center of the bounding box. With high localization accuracy, Pioneer 3-AT overlaps with the center of the predicted box and the aforementioned 3D position can be retrieved easily with the depth information.

At first, by using the inverse transformation of (4):

$$^{C}X = \frac{1}{f_x}(x_{center} - o_x)Z \quad and \quad ^{C}Y = \frac{1}{f_y}(y_{center} - o_y)Z, \tag{5}$$

we can acquire the 3D center coordinates $(^{C}X_{center}, ^{C}Y_{center}, ^{C}Z_{center})$ with respect to the camera frame. But at a second glance, we observe that only the depth(Z) is variable in the above equations. The $\frac{1}{f_x}(x_{center} - o_x)$ and $\frac{1}{f_y}(y_{center} - o_y)$ components remain fixed, since:

- (o_x, o_y, f_x, f_y) are the intrinsic parameters and
- (x_{center}, y_{center}) are pixel coordinates, where $x_{center} \in \{0, width\}$ and $y_{center} \in \{0, height\}$.

Therefore, we calculate these operations only once, at the beginning, with the following function which is located in the visualServo/helpers.py module.

```
def createMap(K, width, height):
    mapX, mapY = [], []

    fx = K[0][0]
    fy = K[1][1]
    cx = K[0][2]
    cy = K[1][2]

    for x in range(width):
        mapX.append((x - cx) * (1.0/fx))
    for y in range(height):
        mapY.append((y - cy) * (1.0/fy))

    return mapX, mapY
```

Then, equation (5) is converted to:

$$^{C}X = mapX[x_{center}] * Z \quad and \quad ^{C}Y = mapY[y_{center}] * Z. \tag{6}$$

The parameter Z here is the sensor's depth measurement for the pixel (x_{center}, y_{center}). However, it is a common practice to use a small kernel to estimate the 3D coordinates of a pixel, since Kinect isn't a perfect sensor. This additional step is captured in the next python function and the implementation lies in the visualServo/helpers.py module:

```
def pixelToCoordinates(depth, x, y, mapX, mapY, xOffset=5, yOffset=5, minPoints=3):
    X, Y, Z = 0, 0, 0
    pointsX, pointsY, pointsZ = [], [], []

    # Scan neighbors #
    for i in range(y - yOffset, y + yOffset):
        for j in range(x - xOffset, x + xOffset):

            currDepth = depth[y][x] / 1000.0
            if currDepth != 0:
                pointsZ.append(currDepth)
                pointsX.append(mapX[j] * currDepth)
                pointsY.append(mapY[i] * currDepth)

    if len(pointsX) < minPoints:
        return 0.0, 0.0, 0.0

    X = sum(pointsX) / len(pointsX)
    Y = sum(pointsY) / len(pointsY)
    Z = sum(pointsZ) / len(pointsZ)

    return X, Y, Z
```

The final step is to transform the 3D coordinates of the center of the box with respect to the Summit-XL coordinate system: $(^{C}X_{center}, ^{C}Y_{center}, ^{C}Z_{center}) \rightarrow (^{R}X_{center}, ^{R}Y_{center}, ^{R}Z_{center})$. If the rotation and the translation of the camera with respect to the Follower is available, then the previous conversion can be computed as follows(see also the cameraToRobot function in the visualServo/helpers.py module):

$$\begin{bmatrix} ^{R}X \\ ^{R}Y \\ ^{R}Z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{12} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} ^{C}X \\ ^{C}Y \\ ^{C}Z \\ 1 \end{bmatrix}$$

For our Scheme, there was no need to find these matrices, as they have been determined in earlier projects. Nevertheless, if someone desires to discover these values, he can follow the next steps:

1. The relative pose of the camera with respect to a chessboard can be computed by the Pose Estimation OpenCV tutorial. At first, three(or more) coordinates (U, V, W) with respect to the chessboard and their corresponding 2D locations in the image plane can be acquired by using the cv.findChessboardCorners() function. Then, if the pose of the camera was available, the 3D points could be transformed with respect to the camera frame:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{12} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

Unfortunately, the extrinsic parameters (r_{ij}, t_i) are unknown, but we can exploit the projected 2D points. By initializing the extrinsics randomly, we can estimate the 2D locations of the 3D points as follows:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

In this way, we can compare the distance between the predicted 2D points and their true values. When the relative pose is perfect, the predicted 2D points will line up onto the image plane with the true 2D coordinates. But when the pose estimate is incorrect, we can minimize the re-projection error by the Levenberg-Marquardt optimization method and after some iterations we end up with the true camera pose with respect to the chessboard: ${}^{P}\xi_{C}$.

2. The chessboard pose with respect to the robot ${}^{R}\xi_{P}$ can be acquired by a laser meter.

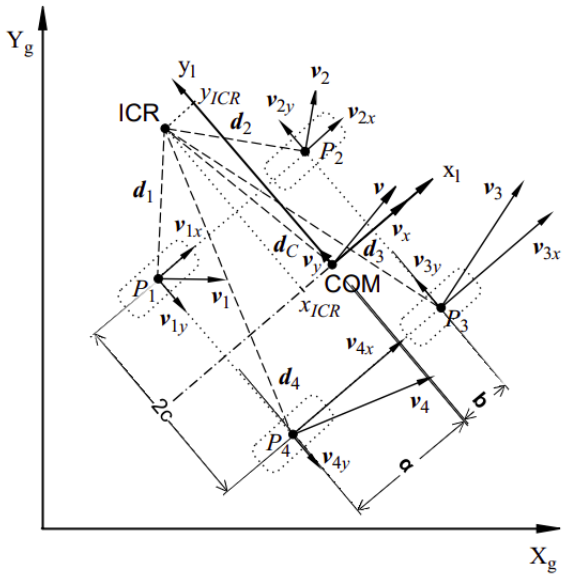
3. Finally, the camera pose with respect to the robot is compounded as follows: ${}^{R}\xi_{C} = {}^{R}\xi_{P} \oplus {}^{P}\xi_{C}$.

4.3 Robot Formation Control

The final stage of the Leader-Follower Scheme is to pick a suitable controller that steers the Summit-XL towards the Pioneer 3-AT.

In the following sections, we begin with the robot model of the Summit-XL and then, we present the controller definition. At last, before testing our system in real experiments with mobile robots, we tune the controller gains via a robotic simulation.

4.3.1 Robot Model



Source:[32]

Figure 4.8: Wheel velocities of Summit-XL

The Summit-XL is a 4-wheel skid-steering mobile robot and its model was presented in a systematic way in [32].

At first, we assume that the longitudinal slip between the wheels and the tangential surface is negligible. Then, it is true that:

$$u_{ix} = r_i \omega_i, \tag{1.1}$$

where u_{ix} is the longitudinal component of velocity u of the i-th wheel and r_i is the effective rolling radius.

Next, we take into consideration all wheels together, as captured in figure 4.8, and we define the radius vectors $d_i = \begin{bmatrix} d_{ix} & d_{iy} \end{bmatrix}^\mathsf{T}$ and $d_C = \begin{bmatrix} d_{Cx} & d_{Cy} \end{bmatrix}^\mathsf{T}$ with respect to the local frame from the instantaneous center of rotation(ICR). Then, we can reason that:

$$\frac{\|u_i\|}{\|d_i\|} = \frac{\|u\|}{\|d_C\|} = |\omega|, \tag{1.2}$$

or, in a more complete form,

$$\frac{u_{ix}}{-d_{iy}} = \frac{u_x}{-d_{Cy}} = \frac{u_{iy}}{d_{ix}} = \frac{u_y}{d_{Cx}} = \omega. \tag{1.3}$$

We can also define the coordinates of ICR in the local frame as follows:

$$ICR = (x_{ICR}, y_{ICR}) = (-d_{xC}, -dyc) \tag{1.4}$$

and then, equation (1.3) can be written as:

$$\frac{u_x}{y_{ICR}} = -\frac{u_y}{x_{ICR}} = \omega. \tag{1.5}$$

Furthermore, d_i vectors satisfy the following relationships:

$$\begin{aligned} d_{1y} &= d_{2y} = d_{Cy} + c, \\ d_{3y} &= d_{4y} = d_{Cy} - c, \\ d_{1x} &= d_{4x} = d_{Cx} - \alpha, \\ d_{2x} &= d_{3x} = d_{Cx} + b \end{aligned} \tag{1.6}$$

and by combining equations (1.3) and (1.6), we can obtain the relationships between the wheel velocities:

$$\begin{aligned} u_L &= u_{1x} = u_{2x}, \\ u_R &= u_{3x} = u_{4x}, \\ u_F &= u_{2y} = u_{3y}, \\ u_B &= u_{1y} = u_{4y}, \end{aligned} \tag{1.7}$$

where u_L, u_R are the longitudinal coordinates of the left and right wheel velocities and u_F, u_B are the lateral coordinates of the velocities of the front and rear wheels, respectively.

To obtain the transformation between the wheel velocities and the velocity of the robot, we can use the equations (1.3)-(1.7) and get:

$$\begin{bmatrix} u_L \\ u_R \\ u_F \\ u_B \end{bmatrix} = \begin{bmatrix} 1 & -c \\ 1 & c \\ 0 & -x_{ICR} + b \\ 0 & -x_{ICR} - \alpha \end{bmatrix} \begin{bmatrix} u_x \\ \omega \end{bmatrix}. \tag{1.8}$$

Assuming that the effective radius is $r_i = r$ for each wheel and according to (1.1) and (1.7), we can define:

$$\omega_\omega = \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix} = \frac{1}{r} \begin{bmatrix} u_L \\ u_R \end{bmatrix}, \tag{1.9}$$

where u_L and u_R are the angular velocities of the left and right wheels, respectively.

Equations (1.8) and (1.9) can describe the relations between the angular wheel velocities and the velocities of the robot by forming a new control input as follows:

$$\eta = \begin{bmatrix} u_x \\ \omega \end{bmatrix} = \frac{1}{r} \begin{bmatrix} \frac{\omega_L + \omega_R}{2} \\ \frac{-\omega_L + \omega_R}{2c} \end{bmatrix}. \tag{1.10}$$

At last, we present the velocity constraint for the Summit-XL(refer also to [33]):

$$u_y + x_{ICR}\dot{\theta} = 0, \tag{1.11}$$

where $q = \begin{bmatrix} X & Y & \theta \end{bmatrix}^\intercal$ is the generalized coordinates vector and $\dot{q} = \begin{bmatrix} \dot{X} & \dot{Y} & \dot{\theta} \end{bmatrix}^\intercal$ denote the generalized velocities vector.

Equation (1.11) is a nonholonomic constraint, as it is not integrable and it can be written in the Pfaffian form as follows:

$$\begin{bmatrix} -\sin\theta & \cos\theta & x_{ICR} \end{bmatrix} \begin{bmatrix} \dot{X} & \dot{Y} & \dot{\theta} \end{bmatrix}^{\intercal} = A(q)\dot{q} = 0. \tag{1.12}$$

It is true that the generalized velocity $\dot{q}$ is always in the null space of A, hence:

$$\dot{q} = S(q)\eta, \tag{1.13}$$

where

$$S^{\intercal}(q)A^{\intercal}(q) = 0 \tag{1.14}$$

and

$$S(q) = \begin{bmatrix} \cos\theta & x_{ICR}\sin\theta \\ \sin\theta & -x_{ICR}\cos\theta \\ 0 & 1 \end{bmatrix}. \tag{1.15}$$

To sum up, equation (1.13) describes the kinematics of the Summit-XL and such a type of robot is underactuated, because $dim(\eta) = 2 < dim(q) = 3$.

4.3.2 Controller

As we mentioned in section 3.1, we desire to find suitable controls (u, ω) so that Summit-XL and Pioneer 3-AT are in d-formation.

To maintain a safe distance d^* from the Pioneer 3-AT and control the Summit-XL's linear velocity, we apply a proportional controller as follows:

$$v^* = K_v\sqrt{\Delta_x^2 + \Delta_y^2} - d^*, \quad K_v > 0 \tag{2.1}$$

where (Δ_x, Δ_y) is the relative position of the Leader with respect to the Follower.

Then, to turn the steering wheels of the Summit-XL towards the Pioneer 3-AT, we use a proportional controller:

$$\omega^* = K_\omega\Delta_{theta}, \quad K_\omega > 0 \tag{2.2}$$

where $\Delta_{theta} = \arctan(\frac{\Delta_y}{\Delta_x})$.

The implementation of the previous controllers lies in the controller/controller.py module. To be precise, the class Controller() contains a method called calculateVelocities(dx, dy) to compute the suitable commands for the relative position (Δ_x, Δ_y) of the Leader.

At last, by using the control inputs (2.1), (2.2) and the equation (1.8), we can calculate the corresponding wheel velocities. However, Robotnik Automation, to soften our work, delivers the Summit-XL with custom ROS packages, such as the movebase. This package can transform the linear and angular velocities to the corresponding wheel velocities by publishing (u, ω) commands to the topic "/summit_xl/robotnik_base_control/cmd_vel". In this way, we save valuable time from building such a basic operation.

4.3.3 Simulation

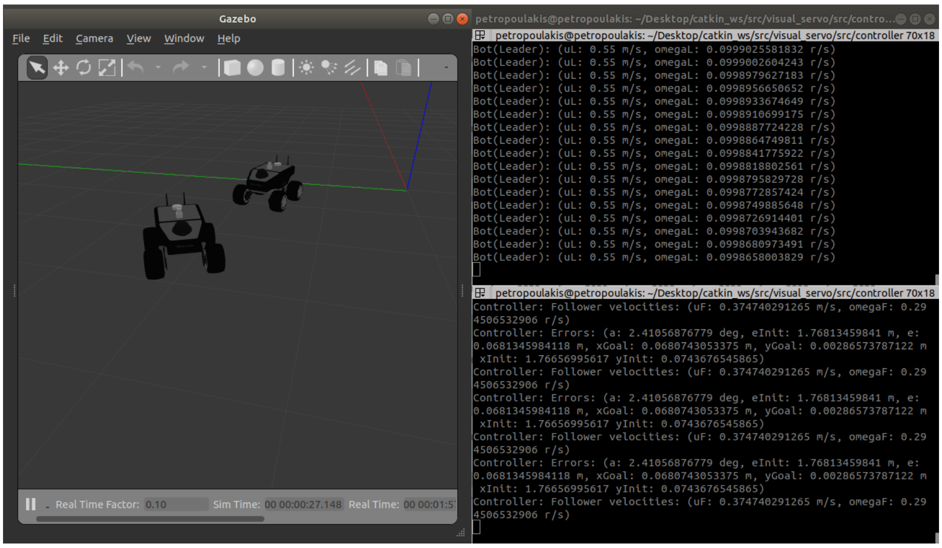


Figure 4.9: Leader-Follower simulation

To tune the gains (K_v, K_ω) so that the Leader remains constantly in the filed of view of the Follower and both robots are in d-formation, we exploited the official ROS simulation of Robotnik called summit_xl_sim(see the figure 4.9 above).

At the end of catkin_ws/.../controller/controller.py module you can follow the instructions to start the simulation. You can also find a class named experiment() that during the simulation steers the Follower by using the odometry of both robots.

We even build a catkin_ws/.../controller/bot.py module to generate random actions(but quite smooth and according to the Pioneer 3-AT velocity ranges) for the Leader robot. To avoid any misunderstanding, we should also note that the summit_xl_sim simulation utilizes a Summit-XL robot for the Leader.

Finally, after a lot of experimentation, the suitable gains for our Leader-Follower Scheme are: $(K_v, K_\omega) = (5.5, 7.0)$.

5. EXPERIMENTAL RESULTS

In this chapter, we validate the proposed system in real life experiments with mobile robots. At first, we present the setup used and the overall communication and then, we report the experimental results and discuss our final thoughts.

5.0.1 Setup

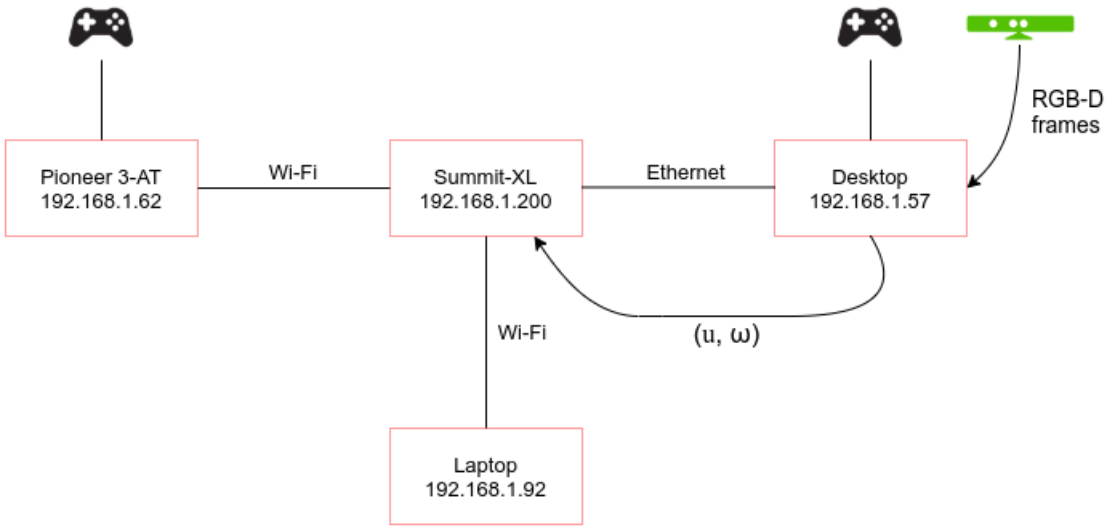


Figure 5.1: Network Architecture

As we mentioned in section 3.2, the Leader-Follower Scheme consists of two robots, one RGB-D Camera, one laptop and one desktop computer. The desktop is placed on top of the Leader and transmits velocities (u, ω) via an Ethernet cable. The Follower and the Laptop are connected to the Summit-XL's network via Wi-Fi as captured in the figure 5.1.

The Pioneer 3-AT is operated by humans with a joystick and the Summit-XL's joystick remained inactive during the experiments, since there wasn't any case of an emergency.

To supply with electricity the desktop computer and the Kinect, we used a power strip which was attached to a plug in our lab.

The Summit-XL and desktop computer should be "in synced" to be able to communicate and exchange velocities (u, ω) in the "/summit_xl/robotnik_base_control/cmd_vel" topic. For that reason, we employed the multimaster_fkie ROS package to establish such a communication.

If you desire to reproduce our experiments, you can find more details in the appendix A.

5.0.2 Experiments

Figure 5.2: Example of a Leader-Follower experiment

The proposed system runs at 19 Hz, but during the experiments, we recorded all the Kinect frames and saved them in an external hard drive and as a consequence, the Scheme was running slower at 14-15 Hz.

Before demonstrating our results, take into consideration that, there weren't any physical obstacles in the experimental location and the Leader's linear velocity was relative stable.

Table 5.1 summarizes the final experiments on the Leader-Follower Scheme. To be more precise, the Kinect Failures column counts the number of frames for which the sensor was unable to return a depth measurement for the center of the bounding box. Additionally, our system records RGB-D frames while the Summit-XL is in motion. We believe that it isn't rational to record frames when the Follower is stopped and remains safe. In appendix B is captured a brief Flowchart of our Scheme and in the media/experiments folder, you can also find additional material such as videos.

Table 5.1: Experimental results about the Leader-Follower Scheme

Experiment	Total Frames	False Detections	Kinect Failures
media/experiments/1	377	03	00
media/experiments/2	362	13	03
media/experiments/3	247	03	00
media/experiments/4	209	01	00
media/experiments/5	316	09	11
media/experiments/6	205	13	00
media/experiments/7	307	00	15
media/experiments/8	193	04	02

As we can observe, the proposed method successfully manages to solve the 1.7-formation of a Leader-Follower Scheme with a low rate of lost frames.

A key conclusion is that, the Kinect occasionally fails to return a depth measurement due to the definition of the Leader's relative position. The reflections of the Infrared Light(IR) on the Leader's ceiling can't be retrieved completely from the IR receiver and as we will explain in the next chapter, an additional improvement would be to process the whole point cloud of the Pioneer 3-AT to reduce even further the Kinect Failures.

6. CONCLUSIONS AND FUTURE WORK

In this work, we examined a Leader-Follower Mobile Robot Scheme using an RGB-D Camera. We employed the MobileNets[1] to detect the Leader in the image frame and then, we processed the depth information to retrieve the Leader's 3D relative position of the center of the bounding box. Finally, we plugged a position based controller to steer the Follower towards the Leader. Multiple experiments on Pioneer 3-AT and Summit-XL showed that our system achieves promising performance and low latency on a low-end GPU card with less than 5% of false detections.

A major conclusion about our system is that, the Leader's position definition entails some drawbacks, because a few reflections on the ceiling can't be retrieved easily from the depth sensor. We believe that, this issue can be cured by processing the whole point cloud.

We familiarized with Open3D[34], a point cloud processing library, and reconstructed a 3D template of the Pioneer 3-AT as captured in figure 6.1. It seems feasible to estimate the real 3D center of the Pioneer 3-AT by aligning point clouds with common registration methods(e.g. [35, 36]). In addition, with these techniques, we can even estimate the relative rotation and employ a different controller to minimize the heading error. In this way, the Summit-XL will remain constantly at the rear of the Pioneer 3-AT. We also notice that, for the aforementioned approach, we might need to apply segmentation methodologies, like Mask R-CNN[37], to isolate completely the Pioneer 3-AT point cloud from the floor or in general, from the background.

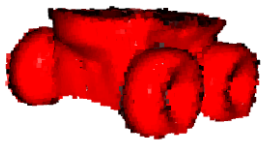


Figure 6.1: Template point cloud of Pioneer 3-AT

As closing remarks, we mention some extra studies and encourage the reader to be puzzled, as the Leader-Follower Schemes include many more extensions and challenges:

1. Deep learning-based 6D pose estimation techniques have recently gained a lot of reputation in robotics. For example, DeepURL[38] estimates directly from an RGB frame the relative pose of a Leader underwater robot. The model predicts bounding boxes and 2D keypoints and then, the pose estimation is performed using 2D-to-3D correspondence pairs and by running a RANSAC-based PnP algorithm. Moreover, the authors suggested a way to train the network by using synthetic labeled data and translating their domain to real underwater images with a CycleGAN[39].

2. A. Saxena et al.[40] trained a Convolutional Neural Network to estimate the relative pose between a desired and a current RGB image. They also validated their system in a real drone by using a position based visual-servo controller.

3. In the same spirit, the authors in [41], trained a CNN to predict the relative pose of the camera from an input image, then plugged a PBVS controller and finally, performed multiple experiments on a 6 DOF Robot.

4. Imagine having multiple Leader-Follower Schemes of identical robots operating at the same place and assigning at the beginning a Leader(e.g. the closest one) to each Follower robot. The main problem here is, how each Follower robot distinguishes his Leader. Object tracking techniques such as SiamMask[42], can solve this problem and we can bring into reality the mentioned scenario.

5. Alex X. Lee et al.[43] at first, trained a multiscale bilinear model to predict the visual features of the next frame given the current image from the robot's camera and the action of the robot. Then, with fitted Q-iteration algorithm learned to estimate directly from the input image actions for steering a Follower robot towards the Leader. At last, they validated their system in a robotic simulation with one drone(follower) and one car(leader).

ABBREVIATIONS - ACRONYMS

RGB-D	Red Green Black and Depth
FPS	Frames Per Second
MED	Mediterranean Conference on Control and Automation
CNN	Convolutional Neural Network
mAP	Mean Average Precision
FC	Fully Connected
NMS	Non Maximum Suppression
IoU	Intersection over Union
SVM	Support Vector Machine
SVMs	Support Vector Machines
ConvNet	Convolutional Network
RoI	Region of Interest
RPN	Region Proposal Network
ReLU	Rectified Linear Unit
ResNet	Residual Network
ROS	Robot Operating System
IMU	Inertial Measurement Unit
ToF	Time of Flight
IR	Infrared Light
SGD	Stochastic Gradient Descent
AR	Average Recall
ICR	Instantaneous Center of Rotation
COM	Center Of Mass
RANSAC	Random Sample Consensus
DOF	Degrees Of Freedom
PBVS	Position Based Visual-Servo

A. INITIATE THE LEADER-FOLLOWER SCHEME

After installing all the required packages(see section 3.2.6), you can follow the next steps to reproduce our experiments on a Leader-Follower Scheme with the accompanied code.

At first, to establish the communication between the topics of the Follower robot and desktop computer by using the multimaster_fkie ROS package:

1. In /etc/hosts file of both computers add the following lines:
   192.168.1.200 summit
   192.168.1.57 cls-mini-pc

2. Connect to Summit-XL(ssh summit@192.168.1.200), open a terminal and type:
   Terminal 1:

```
user@laptop:~$ echo 0 >/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
user@laptop:~$ cat /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
user@laptop:~$ sudo add -net 224.0.0.0 netmask 224.0.0.0 enp1s0
user@laptop:~$ netstat -rn
user@laptop:~$ rosrun master_discovery_fkie master_discovery
```

   †Brief explanation: enable multi cast, add default gateway and discover new nodes.

3. Connect to the desktop computer(ssh csl-mini-pc@192.168.1.57), open 3 terminals and type:
   Terminal 1:

```
user@laptop:~$ echo 0 >/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
user@laptop:~$ cat /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
user@laptop:~$ cd Bsc_thesis/code/catkin_ws/
user@laptop:~$ source devel/setup.bash
user@laptop:~$ rosrun master_discovery_fkie master_discovery
```

   Terminal 2:

```
user@laptop:~$ cd Bsc_thesis/code/catkin_ws/
user@laptop:~$ source devel/setup.bash
user@laptop:~$ rosrun master_sync_fkie master_sync
```

   †Brief explanation: synchronize the local ROS master with the Follower ROS master.
   Terminal 3:

```
user@laptop:~$ rostopic list
```

   †Brief explanation: display the list of topics. Summit-XL topics must be available at this stage.

Finally to start the experiment, type the following commands:

1. Connect to Pioneer 3-AT robot(ssh odroid@192.168.1.62), open 4 terminals and type:

   Terminal 1:

   ```
   user@laptop:~$ roscore
   ```

   Terminal 2:

   ```
   user@laptop:~$ rosrun arduino_mr pioneer_2at_joy_new_pid.py
   ```

   Terminal 3:

   ```
   user@laptop:~$ rosrun arduino_mr pioneer_3dx.py
   ```

   Terminal 4:

   ```
   user@laptop:~$ rosrun teleop_twist_joy teleop.launch
   ```

   †Brief explanation: Start the robot and enable the joystick.

   ∗More information about these routines can be found at BSc thesis of A.Poulias.

2. To run the experiment, connect to the desktop computer and type:

   Terminal 1:

   ```
   user@laptop:~$ cd Bsc_thesis/code/catkin_ws
   user@laptop:~$ source devel/setup.bash
   user@laptop:~$ roslaunch kinect2_bridge kinect2_bridge.launch
   ```

   Terminal 2:

   ```
   user@laptop:~$ cd Bsc_thesis/code/catkin_ws
   user@laptop:~$ source devel/setup.bash
   user@laptop:~$ cd src/visual_servo/src/visualServo/
   user@laptop:~$ python visualServo.py
   ```

   †Brief explanation: Publish RGB-D frames to ROS topics by using the iai_kinect2 package and then, call the visualServo.py module to process these frames, compute the appropriate Summit-XL commands (u, ω) and transmit these commands to the Summit-XL topic "/summit_xl/robotnik_base_control/cmd_vel".

B. FLOWCHART OF THE LEADER-FOLLOWER SCHEME

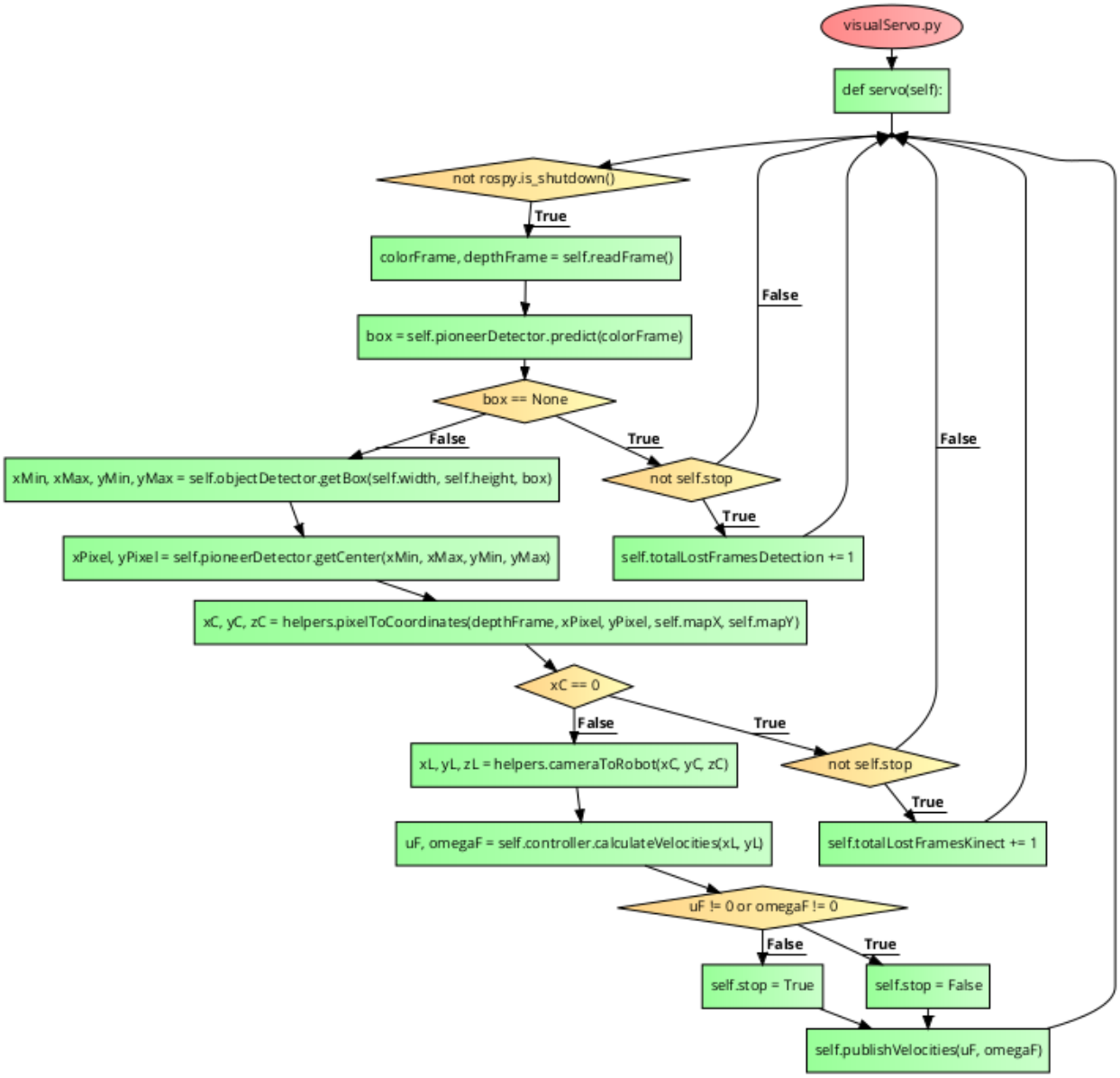


Figure B.1: Brief flowchart of the visualServo.py module

BIBLIOGRAPHY

[1] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” in *CVPR*, 2018.

[2] F. Chaumette and S. Hutchinson, “Visual servo control, Part I: Basic approaches,” in *IEEE Robotics and Automation Magazine*, 2007.

[3] F. Chaumette and S. Hutchinson, “Visual servo control, Part II: Advanced approaches,” in *AISTATS*, 2007.

[4] H. Bay, T. Tuytelaars, and L. V. Gool, “SURF: Speeded up robust features,” in *European Conference on Computer Vision*, 2006.

[5] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An Efficient Alternative to SIFT or SURF,” in *ICCV*, 2011.

[6] R.Girshick, J.Donahue, T.Darrell, and J.Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *CVPR*, 2014.

[7] R. Girshick, “Fast R-CNN,” in *ICCV*, 2015.

[8] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” in *ICCV*, 2015.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *CVPR*, 2016.

[10] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed, “SSD: Single shot multibox detector,” in *European Conference on Computer Vision*, 2016.

[11] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” in *CVPR*, 2017.

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *CVPR*, 2016.

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenk, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” in *CVPR*, 2017.

[14] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The PASCAL Visual Object Classes (VOC) Challenge,” in *IJCV*, 2010.

[15] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, “Selective search for object recognition,” in *IJCV*, 2013.

[16] A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” in *NIPS*, 2012.

[17] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. FeiFei, “ImageNet Large Scale Visual Recognition Competition 2012,” http://www.image-net.org/challenges/LSVRC/2012/.

[18] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan, “Object detection with discriminatively trained part based models,” in *TPAMI*, 2010.

[19] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *ICLR*, 2015.

[20] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” in *NIPS*, 2014.

[21] J. Xue, J. Li, and Y. Gong, “Restructuring of deep neural network acoustic models with singular value decomposition,” in *Interspeech*, 2013.

[22] T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick, “Microsoft COCO: Common Objects in Context,” in *European Conference on Computer Vision*, 2014.

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” in *IJCV*, 2015.

[24] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International Conference on Machine Learning*, 2015.

[25] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller, “Introduction to wordnet: An on-line lexical database,” in *International journal of lexicography*, 1990.

[26] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *AISTATS*, 2010.

[27] R. C. Çalik and M. F. Demirci, “CIFAR-10 Image Classification Using Feature Ensembles,” in *International Conference on Computer Systems and Applications*, 2018.

[28] L. Consolini, F. Morbidi, D. Prattichizzo, and M. Tosques, “Leader-Follower Formation Control of Nonholonomic Mobile Robots with Input Constraints,” in *Automatica*, 2008.

[29] Y. Roh, G. Heo, S. E. Whang, and S. Member, “A Survey on Data Collection for Machine Learning: A Big Data - AI Integration Perspective,” in *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[30] D. P. Kingma and J. L. Ba, “Adam: A Method for Stochastic Optimization,” in *International Conference on Learning Representations*, 2014.

[31] P. Corke, *Robotics, Vision and Control Fundamental Algorithms in MATLAB*. Springer, 2011.

[32] K. Kozłowski and D. Pazderski, “MODELING AND CONTROL OF A 4-WHEEL SKID-STEERING MOBILE ROBOT,” in *International Journal of Applied Mathematics and Computer Science*, 2004.

[33] L. Caracciolo, A. D. Luca, and S. Iannitti, “Trajectory tracking control of a four-wheel differentially driven mobile robot,” in *IEEE Int. Conf. Robotics and Automation, Detroit*, 1999.

[34] Q. Zhou, J. Park, and V. Koltun, “Open3D: A Modern Library for 3D Data Processing,” arXiv:1801.09847, 2018.

[35] S. Rusinkiewicz and M. Levoy, “Efficient variants of the ICP algorithm,” in *Digital Imaging and Modeling*, 2001.

[36] Q. Zhou, J. Park, and V. Koltun, “ORB: An Efficient Alternative to SIFT or SURF,” in *ECCV*, 2016.

[37] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” in *ICCV*, 2017.

[38] B. Joshi, M. Modasshir, T. Manderson, H. Damron, M. Xanthidis, A. Q. Li, I. Rekleitis, and G. Dudek, “DeepURL: Deep Pose Estimation Framework for Underwater Relative Localization,” arXiv:2003.05523, 2020.

[39] J. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks,” in *ICCV*, 2017.

[40] Q. Bateux, E. Marchand, J. Leitner, F. Chaumette, and P. Corke, “Training Deep Neural Networks for Visual Servoing,” in *ICRA*, 2018.

[41] A. Saxena, H. Pandya, G. Kumar, A. Gaud, and K. M. Krishna, “Exploring Convolutional Networks for End-to-End Visual Servoing,” in *ICRA*, 2017.

[42] Q. Wang, L. Zhang, L. Bertinetto, W. Hu, and P. H. Torr, “Fast Online Object Tracking and Segmentation: A Unifying Approach,” in *CVPR*, 2019.

[43] A. X. Lee, S. Levine, and P. Abbeel, “Learning Visual Servoing with Deep Features and Fitted Q-Iteration,” in *ICLR*, 2017.