Visual Localization using Artificial Intelligence

Presented by:

Alesia HAJRULLA
Kristina KUMBRIA

Presented to:

Prof. Pedro CASTILLO

Dr. Jossue CARINO

Table of Contents

ACKNOWLEDGEMENT

First of all, we would like to take this chance and thank the project leaders: Prof. Pedro CASTILLO, and Dr. Jossue CARINO for their none-stop support on this project.

They have accompanied us throughout the semester through weekly meetings. Their involvement in this new project allowed us to move forward serenely and efficiently. However they let us lead the project as we wanted, which was very formative.

CONFIDENTIAL NOTICE

As the project presented in this report is confidential, it is given exclusively to the project leaders who will decide to whom it should be sent.

1. INTRODUCTION

In any robot platforms the position of the robot is one of the most crucial information to be generated, as for example, the location of a vehicle can be used to estimate important data like its speed and acceleration. Even the coordinates of an external object can be employed for tracking, physical manipulation, obstacle avoidance or even navigation in unknown environments.

The Global Navigation Satellite System (GNSS) is the most famous and spread-out system for mobile localization outdoor. The most famous GNSS is the global positioning system (GPS). There are four GNSS systems GPS (US), GLONASS (Russia), Galileo (EU), BeiDou (China),but the most used sensor to estimate position is GPS.

The main shortcomings of these technologies are their high inaccuracy and inability to work in closed spaces. The miniaturization and low cost of camera sensors have made it a viable alternative to use machine vision instead of these Geo-positioning sensors for localization. To localize a robot (in our specific case to determine the pose of the drone), a drone in our specific case study, we can use the data from machine vision techniques with SLAM (Simultaneous Localization and Mapping) techniques.

SLAM is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. We can find many algorithm and methods for this technique but the usage of a single camera introduces several challenges for the SLAM system. It is possible to estimate the bearing of a point in a video frame with the knowledge of the camera model (focal length, distortion parameters), but the distance of the point cannot be measured. That is a problem when we want to add an observed landmark to the map. For that we need more observations of the same landmark from different positions.

1.1. Objectives

There is a triple objective to this project.

Our first objective was to classify different features in a realistic environment using artificial intelligence. A study on the various Machine Learning techniques in machine vision should be provided. This would help to create a more general picture of how Machine learning methods can be used to identify external features in the environment. After the study, our work includes applying the Machine Learning technique that best meets the specific conditions of our case study.

The second objective is the estimation of the relative position of each feature using machine vision algorithms .An application of the technique that has the most accurate results should be done.

Thirdly, all the data acquired from the classification and estimation of the relative position of each feature should be used to determine the relative pose of the monocular camera of the drone.

1.2. Project organization

1.3.1. Team organization

In order to attain all the objectives of this project, on each task presented on the Gant Diagram in Appendix A every team member had a subtask. During our weekly meeting we would present our work, and results, in order to merge our work and find the solution that best meets the demands of our project.

1.3.2. Tasks realised

In the table below is shown a schema of the main tasks realized by each member.

Table 1 **Task Division**

<table>
<tr><td></td><td>Kristina Kumbria</td><td>Alesia Hajrulla</td><td>Kristina Kumbria</td><td>Alesia Hajrulla</td></tr>
<tr><td>ML Algorithms</td><td>Neural Networks</td><td>K - Nearest Neighbors</td><td>Naive Bayesian</td><td>(SVM)</td></tr>
<tr><td>Descriptors and Detectors</td><td>SURF</td><td>HOG</td><td>Brief, ORB</td><td>SIFT</td></tr>
<tr><td rowspan="3">Implementation Stage</td><td colspan="4">Suitable Dataset Research, Familiarizing with Software tools</td></tr>
<tr><td colspan="2">HOG SURF</td><td colspan="2">SIFT ORB</td></tr>
<tr><td colspan="4">Training & testing the dataset using Machine learning algorithm
KNN | SVM</td></tr>
<tr><td>ML</td><td>R-CNN</td><td>Fast R-CNN</td><td>Faster R-CNN</td><td>YOLO</td></tr>
<tr><td>Labeling</td><td colspan="4">Labeling</td></tr>
<tr><td rowspan="2">Mapping</td><td colspan="4">SLAM, FAST-SLAM</td></tr>
<tr><td>Kalman filter</td><td>EKF</td><td>UKF</td><td>Particle filter</td></tr>
</table>

In Appendix A, with detailed set of tasks is presented for each subgroup using Gantt charts. It can also be found the chart done at the very beginning of the project.

1.3.3. Tools used

During the course of the project, several tools were used that helped us better organize the work. We mention some of the most relevant below:

- Python, C++ - The programing languages used to build our code
- OpenCV ,TensorFlow, Scikit-learn – machine learning libraries
- ROS open-source framework that helps researchers and developers build and reuse code between robotics applications

In

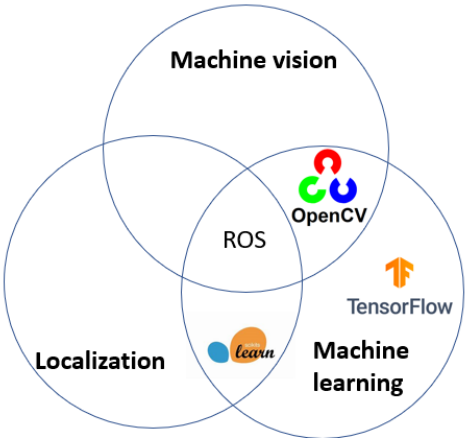


Figure 1 it is shown a graphical schema where every tool that we have used and their reason of usage is presented.

Figure 1: Tools used

2. PLATFORM DESCRIPTION :

2.1. The Pinhole model of the Camera

The pinhole model is very important for our project. It is the model relating the position of a point in the 3D reference frame to its position in the 2D camera frame given the pose of the camera. We will use it in the next parts to estimate the position of the observed objects in the reference frame given the measurement made by the drone (the 2D output of the YOLO code).In the rest of the report, we will also call this pinhole model the measurement model.

The pinhole camera model explains the relationship between a point in the world and the projection on the image plane (image sensor).

If we use a wide-open camera sensor, we will end up with blurry images, because the imaging sensor collects light rays from multiple points on the object at the same location on the sensor. The solution to this problem is to put a barrier in front of the imaging sensor with a tiny hole. The barrier allows only a limited number of light rays to pass through the hole, and reduces the blurriness of the image.

Figure 2. Real image with different aperture sizes

The two most important parameters in a pinhole camera model

1. Focal length: the distance between the pinhole and the image plane.
   It affects the size of the projected image. It affects the camera focus when using lenses.
2. Camera center: The coordinates of the center of the pinhole.

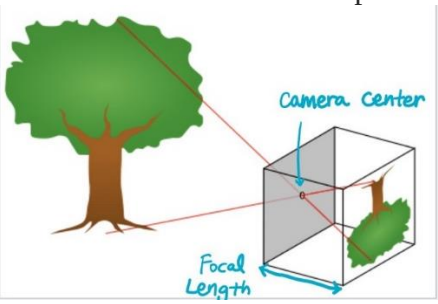


Figure 3. A diagram of a pinhole camera

The pinhole camera model is very straightforward. By knowing the focal length and camera's center, we can mathematically calculate the location where a ray of light that is reflected from an object will strike the image plane.

The focal length and the camera center are the camera intrinsic parameters, K. (K is an industry norm to express the intrinsic matrix.)

Intrinsic parameters

$$K=\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

(C*x, Cy)* : camera center in pixels.(*fx*, *fy*) : Focal length in pixels. $fx = F/px$ $fy = F/py$

F : Focal length in world units (e.g. millimeters.)

In order to project the point in the world frame to the camera image plane, we use Coordnate system transformation!

Light (reflected from the object) travels from the world through the camera aperture (pinhole) to the sensor surface. The projection onto the sensor surface through the aperture results in flipped images. To avoid the flipping confusion, we define a virtual image plane (yellow plane) in front of the camera center.

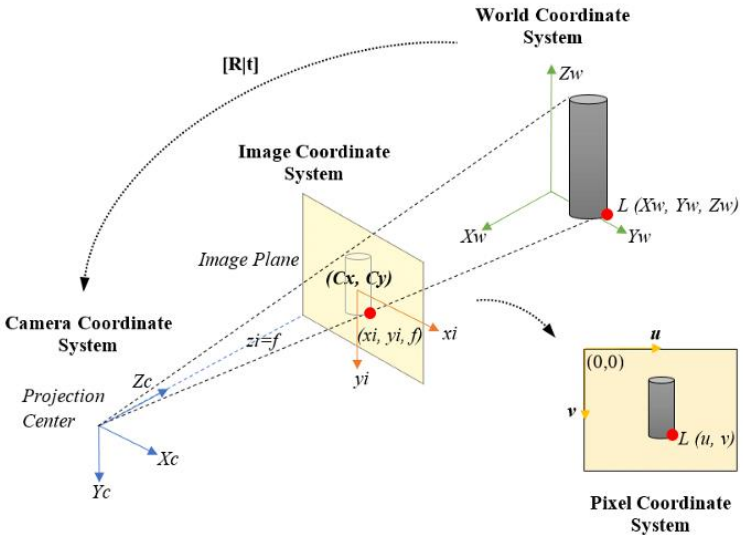


Figure 4. Pinhole model (Projection of coordinate)

We define a 3 by 3 rotation matrix (R) and a 3 by 1 translation vector (t) in order to model any transformation between a world coordinate system and another.
Now we can frame the projection problem (World Coordinates → Image Coordinates) as

1. World coordinates → Camera coordinates
2. Camera coordinates → Image coordinate

We assign
World coordinates as $[X_w, Yw, Zw]^T$
Camera coordinates=$[Xc, Yc, Zc]^T$
Image coordinates=$[x,y,z]^T$

In order to do the calculation, we use the homogeneous coordinate (or projective coordinate) for world coordinates.
So we will have
World coordinates = $[X_w, Yw, Zw,1]^T$

1. Camera coordinates=[R|t]*World coordinates
2. Image coordinates=K* Camera coordinates

If we want to further transform image coordinates to pixel coordinates: Divide x and y by z to get homogeneous coordinates in the image plane.
[x, y, z] -> [u, v, 1] = 1/z * [x, y, z]
Some notes:

a. The rotation matrix (R) and the translation vector (t) are called extrinsic parameters because they are "external" to the camera.
b. The translation vector *t* can be interpreted as the position of the world origin in camera coordinates, and the columns of the rotation matrix *R* represent the directions of the world-axes in camera coordinates. This can be a little confusing to interpret because we are used to thinking in terms of the world coordinates.
c. Usually, multiple sensors (e.g. camera, lidar, radar, etc.) are used for perception in self-driving vehicles. Each sensor comes with its own extrinsic parameters that define the transform from the sensor frame to the vehicle frame.

d. Image coordinate (virtual image plane) [u, v] starts from the top left corner of the virtual image plane. That's why we adjust the pixel locations to the image coordinate frame.

3. OBJECT DETECTION

The first step of this project is to recognize the observed objects from the camera of the drone, in order to track them and get their position in the camera frame, which we will use later to compute its pose. In order to start with this goal it is very important to understand machine vision principles like descriptors and keypoint detectors.

3.1. Descriptors and detectors

In order to start with descriptors and detectors it is important to start with features. A feature is a piece of information which is relevant for solving the computational task related to a certain application. Features may be specific structures in the image such as points, edges or objects. Features may also be the result of a general neighborhood operation or feature detection applied to the image. [1]

The features can be classified into two main categories:

- The features that are in specific locations of the images, such as mountain peaks, building corners, doorways, or interestingly shaped patches of snow. These kinds of localized features are often called keypoint features (or even corners) and are often described by the appearance of patches of pixels surrounding the point location.
- The features that can be matched based on their orientation and local appearance (edge profiles) are called edges and they can also be good indicators of object boundaries and occlusion events in the image sequence.

3.1.1. SIFT

SIFT stands for Scale-Invariant Feature Transform. SIFT is invariance to image scale and rotation. SIFT is quite an involved algorithm. There are mainly four steps involved in the SIFT algorithm. We will see them one-by-one.

- **Scale-space peak selection:** Potential location for finding features.
- **Keypoint Localization:** Accurately locating the feature keypoints.
- **Orientation Assignment:** Assigning orientation to keypoints.
- **Keypoint descriptor:** Describing the keypoints as a high dimensional vector.
- **Keypoint Matching**

Advantages of SIFT

- **Locality:** features are local, so robust to occlusion and clutter (no prior segmentation)
- **Distinctiveness:** individual features can be matched to a large database of objects
- **Quantity:** many features can be generated for even small objects
- **Efficiency:** close to real-time performance

- **Extensibility:** can easily be extended to a wide range of different feature types, with each adding robustness

3.1.2. BRIEF(Binary Robust Independent Elementary Features) [2]

BRIEF(Binary Robust Independent Elementary Features), uses binary strings as an efficient feature point descriptor. BRIEF is very fast both to build and to match.Brief convert image patches into a binary feature vector so that together they can represent an object. Binary features vector also know as binary feature descriptor is a feature vector that only contains 1 and 0. In brief, each keypoint is described by a feature vector which is 128–512 bits string.

It basically works by comparing pixels intensities one to one. More specifically, we define test τ on patch p of size S × S as :

$$\tau(p,x,y) = \begin{cases} 1 if p(x) < p(y) \\ 0 otherwise \end{cases} \quad (1)$$

where p(x) is the pixel intensity in a smoothed version of p at x = (u, v) ⊤. Choosing a set of n_d (x, y)-location pairs uniquely defines a set of binary tests. We take our BRIEF descriptor to be the n_d -dimensional bitstring.

$$f_{n_d} = \sum_{1 \le i \le n_d} 2^{i-1} \tau(p, x_i, y_i) \quad (2)$$

Considering n_d = 128, 256, and 512 and will show in the Results section that these yield good compromises between speed, storage efficiency, and recognition rate.

Advantages of BRIEF are as following.Brief relies on a relatively small number of intensity difference tests to4 represent an image patch as a binary string. Not only is construction and matching for this descriptor much faster than for other state-of-the-art ones, but it also tends to yield higher recognition rates, as long as invariance to large in-plane rotations is not a requirement.

One main main disadvantages is that it is variant to rotation and scale, so it is not the best descriptor to be used in Visual Localisation of UAV.

3.1.3. ORB (Oriented FAST and Rotated BRIEF) [3]

Oriented FAST and Rotated BRIEF (ORB) was developed at OpenCV labs by Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski in 2011, as an efficient and viable alternative to SIFT and SURF. ORB was conceived mainly because SIFT and SURF are patented algorithms. ORB, however, is free to use and offers the following advantages towards BRIEF.

- The addition of a fast and accurate orientation component to FAST.
- The efficient computation of oriented BRIEF features. Analysis of variance and correlation of oriented BRIEF features.
- A learning method for de-correlating BRIEF features under rotational invariance, leading to better performance in nearest-neighbor applications.

3.2. Naive Bayes

The Naive bayes approach is a supervised learning method which is based on a simplistic hypothesis: it assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature. Despite the fact that it is based on a simple hypothesis, its performance is comparable to other supervised learning techniques.

It is also part of a family of algorithms called supervised learning algorithms. Since Naive Bayes is a supervised learning method, it has two distinct stages:

The first stage is the training. This stage uses training data from a set of finite instances of data samples to build a classifier (which will be used in stage 2). This is the so-called supervised learning method—learning from a set of samples and then using this information for the new data classification.

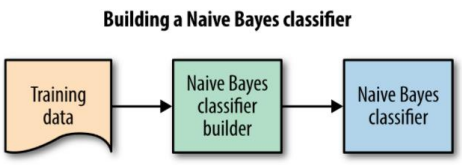


Figure 5 . Stage 1 Training

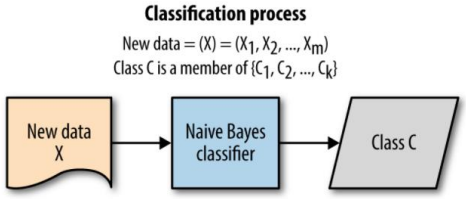


Figure 6. Stage 2: Classification

The second stage is classification. In this stage, we use training data and Bayes's theorem to classify new data in one of the categories identified in stage 1.

This classification algorithm is called Naïve because it assumes each feature, that will be used in order to make the classification to be **independent**, and it assumes that each feature is given the same weighted contribution to the outcome.

Since these assumptions , are not generally true in the real world it is called Naïve. Even though it uses a naïve approach and apparently oversimplified assumptions, it has been proved to work quite well in many complex real-world situations. In 2004, an analysis of the Bayesian classification problem showed that there are sound theoretical reasons for the apparently implausible efficacy of naive Bayes classifiers [4].

It is called Bayesian because it is based on the Bayes Theorem , where E is the Evidence, H is the Hypothesis, and P(H|E) is the conditional probability that H is true with the condition that the evidence is true.

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Specifically, a naive Bayes classifier is based on:

$$P(y|x_1, \dots., x_j) = \frac{P(x_1, \dots., x_j|y)P(y)}{P(x_1, \dots., x_j)}$$

where:

- $P(y \mid x_1, \cdots, x_j)$ is called the *posterior* and is the probability that an observation is class y given the observation's values for the j features, $x_1, \cdots, x_j$.
- $P(x_1, \ldots x_j \mid y)$ is called likelihood and is the *likelihood* of an observation's values for features, $x_1, \ldots, x_j$, given their class, y.
- $P(y)$ is called the *prior* and is our belief for the probability of class y before looking at the data.
- $P(x_1, \ldots, x_j)$ is called the *marginal probability*.

3.3. SVM

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector, and hence algorithm is termed as Support Vector Machine.

The distance between the vectors and the hyperplane is called as margin. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane.

Figure 7.SVM functioning

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.

- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Advantages:

- SVM can use both regression and classification problems.
- SVM can be used on linear and non linear data.
- SVM works well on small datasets.
- Memory efficient algorithm (Use only a subset of the training set for decision function).
- Works well even in cases where the number of dimensions is greater than the number of samples.

Disadvantages:

- SVM is not suited to larger datasets (Training time will be much high).
- SVM does not provide probability estimates. It calculates using expensive 5-fold cross-validation.

3.4. Neural Network

In order to define a neural network we first must introduce two main component of this algorithm.

An artificial neuron takes p inputs {x}, combines them to obtain a single value, and applies an activation function g to the result. The role of the activation function: The initial idea behind the activation function "g" is that it works somehow as a gate and its purpose is to introduce non-linearity into the network.

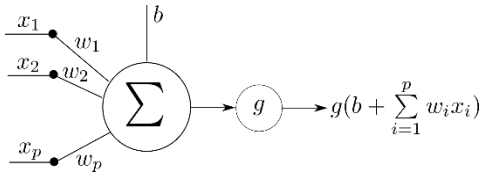


Figure 8 . Artificial neuron

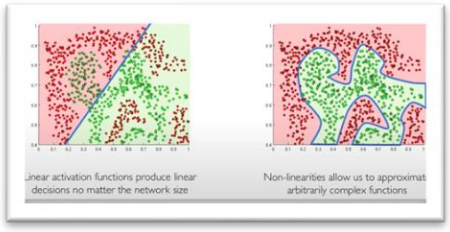


Figure 9. Activation functions

An artificial neural network is a computational graph, where the nodes are artificial neurons, the input layer is the set of neurons without incoming edges and the output layer is the set of neurons without outgoing edges. Any layers other than input and output layers are called hidden layers. Artificial neural networks can be seen as computational graphs processing arrays.

Neural Networks have a lot of potential. However, the parameters θ=(weights, bias) should be set using training techniques

Loss Function: We will define a function that will show us the marge of error our network did by calculating the difference between the proposed answer by the network and the real one using specific techniques.

We choose a loss function "l" and a family "f" of functions from R^p into R, depending on a set of parameters θ, and find the value of θ* that minimizes:

$$\frac{1}{n}\sum_{i=1}^{n} L(F_\theta(x_i), y_i)$$

The choice of the loss function depends on the type of problem and is tightly linked to the application.

we will see an example of a classical loss function: the squared error loss.

A square error loss Function has the following Form

$$L)= \left(y_i - F_\theta(x_i)\right)^2$$

To minimize this loss we will use the gradient descent:

first the loss is calculated $L(\theta_t = L$, then the gradient $\boldsymbol{\nabla}\ L(\theta_t$ is computed and finally the parameters are updated

$$\theta_{t+1} = \theta_t - \eta\ \boldsymbol{\nabla}\ L(\theta_t$$

where η will be the learning rate of our model.

Since we will use the CNN for image processing, our network will directly take as input the image pixels and it is supposed to build its own features.

More specifically we will work with the classification problem:

Input: image

Output: class (y) chosen from a set of labels

Classically, an image is a matrix of values belonging to [0, … ,255] (grey level images) or to $[0, \dots, 255]^3$ (color images). More generally, an image is a q-dimensional array of values belonging to R^d

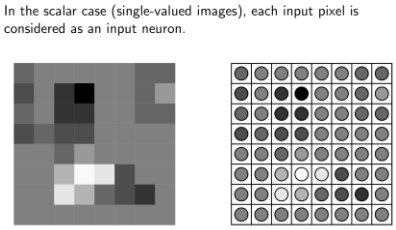


Figure 10. Image as a q-dimensional array of values

- A small image easily contains 100 000 pixels
- The number of parameters between two layers of that size is10^5 (10^5 + 1)!
- The fully connected neural network approach is only possible for very small images

Many successful architectures, especially for image classification, follow the same pattern:

1. one or several convolutional layers, with increasing depth.
2. Applying max pooling to extract the most important features calculated
3. a few fully connected layers that leads to the output

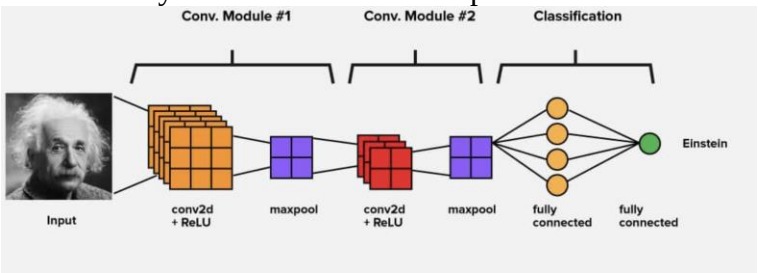


Figure 11. The Neural Networks architecture

Convolution neural network go through a well-defined algorithm that include features extraction and classification to finally be able to sort out specific details in an image given as input

3.5. R-CNN

Instead of working on a massive number of regions, the R-CNN algorithm proposes a bunch of boxes in the image and checks if any of these boxes contain any object. R-CNN uses selective search to extract these boxes from an image (these boxes are called regions).

There are basically four regions that form an object: varying scales, colors, textures, and enclosure. Selective search identifies these patterns in the image and based on that, proposes various regions.

Below is a summary of the steps followed in R-CNN to detect objects:

1. We first take a pre-trained convolutional neural network.
2. Then, this model is retrained. We train the last layer of the network based on the number of classes that need to be detected.

3. The third step is to get the Region of Interest for each image. We then reshape all these regions so that they can match the CNN input size.
4. After getting the regions, we train SVM to classify objects and background. For each class, we train one binary SVM.
5. Finally, we train a linear regression model to generate tighter bounding boxes for each identified object in the image.

Problems with R-CNN

- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.
- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- Training an RCNN model is expensive and slow thanks to the below steps:
    - Extracting 2,000 regions for each image based on selective search
    - Extracting features using CNN for every image region. Suppose we have N images, then the number of CNN features will be N*2,000

3.6. Fast R-CNN

Instead of running a CNN 2,000 times per image, we can run it just once per image and get all the regions of interest (regions containing some object).

In Fast R-CNN, we feed the input image to the CNN, which in turn generates the convolutional feature maps. Using these maps, the regions of proposals are extracted. We then use a RoI pooling layer to reshape all the proposed regions into a fixed size, so that it can be fed into a fully connected network.

Let’s break this down into steps to simplify the concept:

1. As with the R-CNN, we take an image as an input.
2. This image is passed to a ConvNet which in turns generates the Regions of Interest.
3. A RoI pooling layer is applied on all of these regions to reshape them as per the input of the ConvNet. Then, each region is passed on to a fully connected network.
4. A softmax layer is used on top of the fully connected network to output classes. Along with the softmax layer, a linear regression layer is also used parallely to output bounding box coordinates for predicted classes.

So, instead of using three different models (like in R-CNN), Fast R-CNN uses a single model which extracts features from the regions, divides them into different classes, and returns the boundary boxes for the identified classes simultaneously.

Problems with Fast R-CNN

- It also uses selective search as a proposal method to find the Regions of Interest, which is a slow and time consuming process.
- It takes around 2 seconds per image to detect objects.

3.7. Faster R-CNN

Faster R-CNN is the modified version of Fast R-CNN. The major difference between them is that Fast R-CNN uses selective search for generating Regions of Interest, while Faster R-CNN uses "Region Proposal Network", aka RPN. RPN takes image feature maps as an input and generates a set of object proposals, each with an objectness score as output.

The below steps are typically followed in a Faster R-CNN approach:

1. We take an image as input and pass it to the ConvNet which returns the feature map for that image.
2. Region proposal network is applied on these feature maps. This returns the object proposals along with their objectness score.
3. A RoI pooling layer is applied on these proposals to bring down all the proposals to the same size.
4. Finally, the proposals are passed to a fully connected layer which has a softmax layer and a linear regression layer at its top, to classify and output the bounding boxes for objects.

Problems with Faster R-CNN

The network does not look at the complete image in one go, but focuses on parts of the image sequentially. This creates two complications:

- The algorithm requires many passes through a single image to extract all the objects
- As there are different systems working one after the other, the performance of the systems further ahead depends on how the previous systems performed

3.8. YOLO

YOLO is an abbreviation for the term 'You Only Look Once'. This is an algorithm that detects and recognizes various objects in a picture (in real-time). Object detection in YOLO is done as a regression problem and provides the class probabilities of the detected images.

YOLO is refreshingly simple. A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection.

YOLO is extremely fast. Since we frame detection as a regression problem we don't need a complex pipeline. We simply run our neural network on a new image at test time to predict

detections. Our base network runs at 45 frames per second with no batch processing on a Titan X GPU and a fast version runs at more than 150 fps. This means we can process streaming video in real-time with less than 25 milliseconds of latency. Furthermore, YOLO achieves more than twice the mean average precision of other real-time systems.

YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN, a top detection method [5], mistakes background patches in an image for objects because it can't see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN.

YOLO learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs. YOLO still lags behind state-of-the-art detection systems in accuracy. While it can quickly identify objects in images it struggles to precisely localize some objects, especially small ones. We examine these tradeoffs further in our experiments.

3.8.1 How does it work?

To understand the YOLO algorithm, first we need to understand what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

- Center of the box (bx, by)
- Width (bw)
- Height (bh)
- Value c corresponding to the class of an object

Along with that we predict a real number pc, which is the probability that there is an object in the bounding box.

First, the image is divided into various grids. Each grid has a dimension of S x S, typically 19x19. The following image shows how an input image is divided into grids.

An Object is considered to lie in a specific cell only if the center co-ordinates of the anchor box lie in that cell. Due to this property the center co-ordinates are always calculated relative to the cell whereas the height and width are calculated relative to the whole Image size. During the one pass of forwards propagation, YOLO determines the probability that the cell contains a certain class. The equation for the same is :

$$score_{c,i} = p_c \times c_i$$

The class with the maximum probability is chosen and assigned to that particular grid cell. Similar process happens for all the grid cells present in the image. After computing the above class probabilities, the image may look like this :

Figure 12. Grid division& maximum probability applied

This shows the before and after of predicting the class probabilities for each grid cell. After predicting the class probabilities, the next step is Non-max suppression, it helps the algorithm to get rid of the unnecessary anchor boxes, like you can see that in the figure below, there are numerous anchor boxes calculated based on the class probabilities.

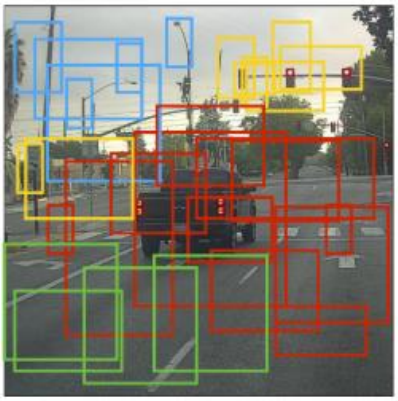


Figure 13. Anchor Boxes

To resolve this problem Non-max suppression eliminates the bounding boxes that are very close by preforming the IoU (Intersection over Union) with the one having the highest class probability among them.

It calculates the value of IoU for all the bounding boxes respective to the one having the highest class probability, it then rejects the bounding boxes whose value of IoU is greater than a threshold. It signifies that those two bounding boxes are covering the same object but the other one has a low probabilty for the same, thus it is eliminated.

Once done, algorithm finds the bunding box with next highest class probabilities and does the same process, it is done until we are left with all the different bounding boxes.

Figure 14. Before and After of Non-max suppression

After this, almost all of our work is done, the algorithm finally outputs the required vector showing the details of the bounding box of the respective class. The overall architecture of the algorithm can be viewed below :

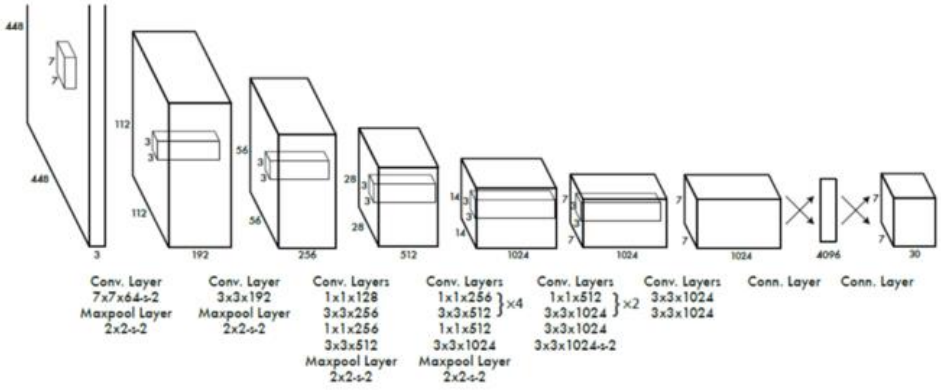


Figure 15. YOLO Architecture [6]

Also, the most important parameter of the Algorithm, its Loss function is shown below. YOLO simultaneously learns about all the four parameters it predicts.

3.8.2 Modification of YOLO (The Labeling algorithm)

To track each object, we need to "improve" the YOLO algorithm. We need to track the bounding box corresponding to the object in order to extract their position on each frame.

To this end, we created a "labelling algorithm" which assigns every newly detected object a number corresponding to its instance. From one frame to another, the algorithm performs a comparison between the newly detected bounding boxes and the previous ones based on the distance between their centers, the average value on each RGB channel (that can be helpful to distinguish two objects that are close but of different colors), a ratio between their respective height and width and their category of course.

We also added a “landmark extraction” feature, that creates a table containing the different objects and their position observed on each frame, which will be used in a following part of the project.

3.9. Random forest

The main idea behind Random Forest machine learning models is to optimize many random small decision trees on the data, and then doing a majority voting of the results of these sub-models in order to have the final prediction, these kind of models have proven to be robust in the case of imbalanced labels.

It works in four steps:

1. Select random samples from a given dataset.
2. Construct a decision tree for each sample and get a prediction result from each decision tree.
3. Perform a vote for each predicted result.
4. Select the prediction result with the most votes as the final prediction.

4. Localization of the drone

To estimate the localization of our drone, we had two solutions. The first one is to localize the drone in an already known environment (we already know the environment and the objects in it, their label and their position in the environment). This would mean that we would train the YOLO algorithm to detect individually the objects that will be in the environment of the drone.

It would be able to differentiate two cars between each other for example, which is nice in order to know exactly what object is observed and to use its position, but this would mean that the use of the drone would be limited to a certain zone/room and we would have to measure the exact position of each object in the zone. Also it would require to build a big dataset with a lot of different views of each object in order to train the YOLO algorithm to recognize them. This solution would therefore be very time-consuming.

The other solution that we used is to implement a SLAM algorithm (Self Localization and Mapping) so the drone can localize itself anywhere.
This allows us to directly use the YOLO algorithm without re-training it as it was trained with the COCO dataset, which can recognize 80 different everyday objects. But this means that we have to create an algorithm that would differentiate two objects of the same category and track them (ex : this is car n°1, this is car n°2).

We will also have to determine their position at the same time as that of the robot, and this is why we will need the SLAM algorithm.

4.1. SLAM

Visual simultaneous localization and mapping, refers to the process of calculating the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment. The process uses only visual inputs from the camera, process the image data to build a map of an indoor environment and estimate the trajectory.

The ORB-SLAM pipeline includes:
-Map Initialization: ORB-SLAM starts by initializing the map of 3-D points from two video frames. The 3-D points and relative camera pose are computed using triangulation based on 2-D ORB feature correspondences.
-Tracking: Once a map is initialized, for each new frame, the camera pose is estimated by matching features in the current frame to features in the last keyframe. The estimated camera pose is refined by tracking the local map.
-Local Mapping**:** The current frame is used to create new 3-D map points if it is identified as a key frame. At this stage, bundle adjustment is used to minimize reprojection errors by adjusting the camera pose and 3-D points.

-Loop Closure: Loops are detected for each key frame by comparing it against all previous keyframes using the bag-of-features approach. Once a loop closure is detected, the pose graph is optimized to refine the camera poses of all the keyframes.

FastSLAM is based on the important observation that the posterior can be factored.
This factorization is exact and universal in SLAM problems. It states that if one knew the path of the vehicle, the landmark positions could be estimated independently from each other. In practice one does not know the vehicle's path. Nevertheless, the independence makes it possible to factor the posterior into a term that estimates the probability of each path, and terms that estimate the position of the landmarks, conditioned on each path.
FastSLAM samples the path using a particle filter. Each particle has attached its own map, consisting of extended Kalman filters

Each update in FastSLAM begins with sampling new poses based on the most recent motion command u_t.Next, FastSLAM updates the estimate of the observed landmark (x), according to the following posterior. This posterior takes the measurement y_t into consideration.In a final step, FastSLAM corrects for the fact that the pose sample x_t has been generated without consideration of the most recent measurement. It does so by resampling the particles.

In FastSLAM both the pose and the positions are subject to measurement noises and some approximations on the model used to model the drone (which is here its kinematic model). We thus have to use techniques to compensate these noises in order to get more accurate results :

4.1.1. Extended Kalman Filter

The Kalman Filter is a very rare algorithm, in that it is one of the few that are provably optimal, At its core, it propagates a state characterized by a Gaussian distribution using linear transition functions in an optimal way. Since it is optimal, it has remained relative-ly unchanged since it

was first introduced, but has received many extensions to apply it to more than just linear Gaussian systems. In this section we sketch the Kalman Filter algorithm, and analyze its computational complexity in the time domain.

Since the Kalman Filter is a Bayesian filter, we will use Bayes' law to estimate an unobservable state of a given system using observable data. They do this by propagating the posterior probability density function of the state using a transition model. First we will introduce the Bayesian equation

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

We first must note the Kalman Filter comes with several assumptions:

1.The state transition is linear in the form

$$x_{t+1} = Ax_t + Bu_t + w_k$$

where x_t is the state, u_t is the control, and w_k is added Gaussian noise.

2. The measurement is linear in the form

$$y_t = Cx_t + v_k$$

where x_t is the observation, and v_k is the added Gaussian noise.

3. The system is continuous.

While these assumptions restrict the applicability of the Kalman Filter, we will show later they also ensure its optimality. The algorithm can be divided into two distinct phases: a time update phase and a measurement update phase

*Time Update phase.*In the time update phase, the state is projected forward. However, we also must propagate the uncertainty in the state forward. Since the state is a Gaussian distribution, and is fully parameterized by a mean s_t and covariance P_t, we can update the covariance.

$$P_{t+1}^{-} = AP_tA^T + Q$$

Here, A is the same matrix used to propagate the state mean, and Q is random Gaussian noise
After the time update phase, the original Gaussian characterized by X_t and P_t is a new Gaussian, now characterized by x_{t+1} and P_{t+1}

$$\hat{x}_{t+1}^{-} = A\hat{x}_t + Bu_t$$
$$P_{t+1}^{-} = AP_tA^T + Q$$

.

This concludes the time update phase, and represents the prediction step of the algorithm.

Measurement Update. The measurement update phase is the correction step of the Kalman Filter, wherein a measurement of an observable variable is made and fused with the prior distribution to estimate the posterior. First, we make a measurement of the system using our linear measurement model. After the measurement is made, we form what is known as the Kalman Gain, this is the key step of the Kalman Filter.

$$K = PC^T(CPC^T + Q)^{-1}$$

Now we are ready to calculate the posterior distribution

$$\hat{x}_t = \hat{x}_t^- + K(y_t - C\hat{x}_t^-)$$
$$P_t = (I - KC)P_t^-$$

Here, and fully parameterize the posterior distribution. The preceding steps represent a single iteration of the Kalman filter. This output is then used as input to a subsequent observation, along with a new control and observation.

One major limitation of the Kalman Filter is the strict set of problems to which it applies: problems with linear state transition and linear measurements with added Gaussian noise. While many problems can be modeled this way, it would be nice to apply the Kalman Filter to other problems, due to its optimality.

The Extended Kalman Filter was invented just for this purpose. It works through a process of linearization, where the nonlinear transition and observation functions are approximated by a Taylor Series expansion.

With the Extended Kalman Filter, we can no longer assume a linear process update or observation model

$$\mathbf{x}_t = \mathbf{f}(\mathbf{x}_{t-1}, u_t)$$
$$\hat{y}_t = \mathbf{g}(\mathbf{x}_{t-1})$$

Since the Kalman Filter works on only linear inputs, we must linearize these functions to propagate the state covariance matrix forward. We do this by approximating the function as a line tangent to the actual function at the mean value. This line is found by expand- ing the nonlinear functions in a Taylor Series around the mean, and taking the first order approximation.

Thus, we need to linearize both equations by taking the gradient of each with respect to the state xt :

$$\mathbf{F}_t = \nabla f(\mathbf{x}_{t-1}, u_t)$$
$$\mathbf{G}_t = \nabla \mathbf{g}(\mathbf{x}_{t-1})$$

Thus we now follow the same predictor-corrector form of the original Kalman Filter in two distinct phases.

Time Update.First, we propagate the mean forward to find the mean of our prior distribution. Next, we have to propagate the covariance forward to be the covariance convolved with the appropriate Jacobian. In this case we use F_t.

$$P_{t+1}^- = F_t P_{t-1} F_t^T + Q$$

Here, Q is again zero mean Gaussian noise of the process model. These two equations provide us with a fully parameterized Gaussian prior. We now move to the measurement update phase.

Measurement Update. The measurement update phase is largely the same as in the Kalman filter, the Kalman Gain is calculated using a Jacobian. This has the unfortunate consequence that Kalman Gain can no longer be considered optimal, and thus the Extended Kalman Filter cannot be the best possible algo-rithm for filtering nonlinear systems.

$$\hat{y} = (y_t - g(x_t))$$

With this last equation, we can estimate the posterior in the standard Kalman Filter way.

4.1.2. Unscented Kalman Filter

When the state evolution and observation models are highly nonlinear, the Extended Kalman Filter can give practically poor performance because the covariance is propagated through linearization. The Unscented Kalman Filter uses a deterministic sampling technique known as the unscented transformation to pick a minimal set of sigma points around the mean. The sigma points are then propagated through the nonlinear functions, from which a new mean and covariance estimate are then formed.
This technique removes the calculation of Jacobian Matrices which is sometimes not possible if the functions are not differentiable.

The Kalman filter gain is given by the same formula K= , we use sigma points to compute and .
And for the update step we apply these formula:

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K(y_t - \hat{y})$$
$$P_{t|t} = P_{t|t-1} - KP_yK^T$$

After having done the update, we have an estimate and a covariance matrix which allows us to compute new sigma points.
For i=1:N , $x^i=f(x^i)$
And so

$$\hat{x}_{t+1|t} = \sum_{i=1}^{N} w^i x^i$$

We compute their variance and add the variance of the model error:

$$P_{t+1|t} = \sum_{i=1}^{N} w^i (x^i - \hat{x})(x^i - \hat{x})^T + Q$$

4.1.3. Particle Filter

We now turn to a nonparametric filter, with a unique approach to calculating the posterior distribution: the Particle Filter. As its name suggests, it uses a set of hypotheses called particles as guesses for the true configuration of the state.The particle filter is different from previous filters in that it is not limited by linear models or Gaussian noise. This flexibility is due to how it represents the probability density function as a set of samples known as particles. Each sample is taken from a proposal distribution, and weighted according to how well it matches a target

distribution. After weighting, the particle distribution does not match the posterior, so we carry out the key step in the Particle Filter algorithm: importance sampling.

The first step in the particle filter is to go through every particle and sample from a proposal distribution.Just as the Bayesian Filters we looked at in previous sections, the Particle Filter is a recursive algorithm, so we therefore sample the current state using the previous state. The same control input u_t as the previous methods is used to propagate the state forward. Since the transition function is noisy, each particle goes through a different
transition, which adds variety to the particle set.
Note that we do not explicitly state how to obtain this sample. This is dependent on the exact system, and is one of the requirements of the Particle Filter, that a proposal distribution must be available to sample from. In many applications, this is readily available. If not the case, it sometimes suffices to sample from a Gaussian distribution with appropriate mean and covariance.
Next each particle is weighted based on how well it matches the posterior distribution.This is equivalent to the measurement update phase in the Kalman Filter, where the observation is incorporated into the belief. There are many different methods to weight particles.When we have completed these two steps, we are finally free to add the new weighted particle to a temporary particle set.
Also known as importance sampling, the resample step uses the newly generated temporary particle set to generate the final posterior distribution. Just as with weighting the particles, there are many ways to accomplish importance sampling, but doing this incorrectly can lead to the wrong posterior distribution. Essentially, the resampling step reduces variance in the particle set, which decreases the accuracy of the posterior approximation. In general, we choose the sample particles by replacing the frequency proportional to their weight. After the resampling step is complete, the new particle set should match the target distribution. This particle set is then used as input to a subsequent iteration of the particle filter algorithm.

4.1.4. Least Square Method

The basic idea behind LMS filter is to approach the optimum filter weights by updating the filter weights in a manner to converge to the optimum filter weight.
This is based on the gradient descent algorithm. The algorithm starts by assuming small weights (zero in most cases) and, at each step, by finding the gradient of the mean square error, the weights are updated.
That is, if the gradient is positive, it implies the error would keep increasing positively if the same weight is used for further iterations, which means we need to reduce the weights. In the same way, if the gradient is negative, we need to increase the weights.
The weight update equation is

$$W_{n+1} = W_n - \mu \nabla \varepsilon[n]$$

where $\boldsymbol{\varepsilon}$ represents the mean-square error and $\boldsymbol{\mu}$ is a convergence coefficient.
The mean-square error as a function of filter weights is a quadratic function which means it has only one extremum, that minimizes the mean-square error, which is the optimal weight. The LMS thus approaches towards this optimal weights by ascending or descending down the mean-square-error versus filter weight curve.

4.2. Argumentation of the choices of the localization algorithms

In order to estimate the position of the drone, we will use a particle filter. It consists in generating a given amount of particles, each one representing a possible pose of the drone and having its own map (the list of the observed landmarks).
Each particle moves in its own way following the kinematic model of the drone (with some noise) and thanks to the pinhole model, computes the corresponding position of the landmarks it should see.

From the measurement made by the drone, each particle is then assigned a weight, the weight corresponding to how close its pose seems to be from the real drone's pose.The estimated pose of the drone then corresponds to the weighted average of the poses of the particles.

In order to estimate the position of the landmarks, the book introducing FastSLAM uses the EKF : Extended Kalman Filter, a Kalman filter used when the measurement model or the system's model are not linear. It represents the variables as Gaussian distributions whose means are their estimated value and whose covariance matrices correspond to the accuracy of the estimation.

It is our case here but we cannot use it as it implies to compute the Jacobian matrix of both these models and the one of the measurement model would be hard to compute. To solve this issue we decided to use an Unscented Kalman Filter, which allows us to estimate the position of the particles without these Jacobians.
But while testing our SLAM code, we were encountering an error noting that the covariance matrix of the landmarks was not positive definite, which should be the case. We concluded that it might be due to the fact the initial positions of the landmarks could be too far from the actual one and this is why the UKF couldn't converge as it relies on a Gaussian distribution.
We then tried to use a particle filter on each landmark (as explained before, it means assigning a landmark several possible positions and converging to the most probable one). But we observed that the estimated position of the landmark was diverging very fast and reaching very high and outlier values (see figures 1 and 2).

This is the point we have reached at the end of our project. Unfortunately there might be an error in the SLAM algorithm we didn't find.

5. Implementation of the code

5.1. Requirements

For the entire project, we used the Anaconda distribution with Python 3.8.8.

The codes we used and created required common libraries that can be installed using pip or conda, excepted for YOLOv5 that requires the use of CUDA, allowing a parallel computation using the GPU, and a version of Pytorch that is compatible with CUDA. These special installations are explained very well in the following tutorial : [7]

5.2. Explanation & How to compile/use the example

YOLO : YOLO is a real-time detection and classification algorithm. In our project we used its last version called YOLOv5, that is available through this link: https://github.com/ultralytics/yolov5, but that we modified so in addition to this report is included the modified YOLO.

Here is the list of the changes we did on the original YOLOv5 code :

- detect.py : **modified** : This code is the "main" code of YOLO, it is meant to be launched from the terminal but we changed this part so we can launch it directly from our IDE, which is way more convenient for the tests.

  detect.py contains the loop performing the detection over each frame, we wrote a few additional lines to include the codes we made on our own.

- labelling.py : **new (used)** : this code is the one that labels the bounding boxes detected by YOLO and tracks them from one frame to another. To check if it is working, you can check the windows opened when launching 'detect.py', in addition to the regular output images of YOLO, another window displays the tracking performed on these boxes and assigns them number corresponding to their instances. It might be improved by modifying the parameters used in the functions 'compareLabBox' and 'outOfBorder' that are the criterias that have to be respected by two successive bounding boxes in order to be considered as being the same object.

- landmarks_extraction.py : **new (used)** : this code is called by 'detect.py' and calls directly 'labelling.py'. It is the one allowing us to create a dictionary containing the position of the centers of the bounding boxes and their label on each frame (the keys of the dictionary corresponding to the frame's position). It is the code containing the function that displays the image with the labelled boxes.

  As it is meant to be chained with the SLAM algorithm and used in real time (what we didn't have the time to do), the table is not stored after the run but it can be easily implemented if required.

- track.py : **new (not used)** : this code was our initial attempt for the tracking of the bounding boxes. It uses computer vision methods to track them but it is not optimal at all and slows the algorithm so much that it can't be used in real-time anymore. This code is no longer used but we left it to remember this method has been tried.

We used the default parameters to call 'detect.py', we just had to indicate the global path to the source image/video.

Also, we used the original weights of the CNN of YOLO that was trained with the COCO dataset, which was useful for us. A tutorial to learn how to train the YOLO algorithm can be found by following this link : https://blog.paperspace.com/train-yolov5-custom-data/

If required, you can also find a tutorial on how to install YOLOv5 and how to use it by following this link : https://wandb.ai/onlineinference/YOLO/reports/YOLOv5-Object-Detection-on-Windows-Step-By-Step-Tutorial---VmlldzoxMDQwNzk4#what-is-yolo?

KITTI dataset :

We decided not to use the KITTI dataset for the localization but a possibility was to train YOLO to detect specific individual objects so we learnt how to train YOLO with KITTI.

Files to train YOLO with the KITTI 2D detection dataset

For our tests, we tried to train YOLO with the KITTI 2D detection dataset, which is pretty easy to use.

To do so, we followed this tutorial : https://blog.paperspace.com/train-yolov5-custom-data/

The codes we used and the files required we the training are contained in the folder Train_KITTI attached with the report. This folder has to be copied and pasted into the YOLO folder.

This folder also contains two code we created :

- convert_labels_kitti2yolo.py : this code converts the labels of the KITTI dataset in a YOLO compatible format. The global paths to the source labels and images have to be changed and also the one to the folder where the news labels have to be stored. In order to verify if this code worked well, you can check that a new file has been created in the target folder for each original label file, and in addition you can use the following code.
- test_labels_kitti.py : this code can be used to test that the labels we will use for the training correspond to the YOLO format. You simply need to give the path to an image from the dataset and to the corresponding label, and the code should display the image with the correct bounding boxes and labels if everything is in order.

We didn't include the dataset as it is way too heavy but in order to use it, a folder named KITTI has to be created in the YOLO folder. The images have to been stored in a folder named KITTI/images/train for the training images and KITTI/images/test for the test ones. The labels have to follow the same principle, for example the training labels must be stored in KITTI/labels/train.

(Please note that the paths to the images can be modified in the file named Train_KITTI/kitti.yaml, but the labels will still have to be found by replace 'images' by 'labels' in the path).

The training can be performed typing the following command in the terminal :

!python train.py --freeze 10 --cfg yolov5s.yaml --hyp Train_KITTI/hyp.yaml --batch 32 --epochs 10 --data Train_KITTI/kitti.yaml --weights yolov5s.pt --workers 16 --name yolo_road_det

As you can see, the parameters we used are low because the training requires a lot of computation and we couldn't train YOLO with higher parameters. Also, we used the function '--

freeze 10' that kept the backbone of the pretrained YOLO on the COCO dataset and allowed us to save a lot of computation and time in exchange of a slight loss of accuracy.

Tool to read the raw Kitti dataset

In this report, we decided to include the code 'extract_pose_kitti.py' that reads the file called 'tracklets_labels.xml' from the part of the raw Kitti dataset called '2011_09_26_drive_0001'. You can find this dataset following this link : http://www.cvlibs.net/datasets/kitti/raw_data.php

The file 'tracklets_labels.xml' contains all the informations about the objects observed in the dataset : their label, their dimensions, pose, the frame they can be seen at, etc.
Similarly to 'landmarks_extraction.py', the code reads the file and builds a dictionary containing each object and its informations that is seen on every frame of the video.

We didn't use but we included it because it might be useful for further works so we think it is worth mentioning it.

SLAM algorithm :

Based on the 'minimum working example' codes (that you can find in the 'Codes/Minimum working examples' folder) that Prof. Cariño Escobar gave us, we created two codes adapted to our project. Both these codes were giving wrong results for the moment and must be completed but they seem to be promising :

- test_slam.py : This is the code performing the FastSLAM method for the localization of the drone and the mapping of the observed objects (that are virtual points for the moment).
  This code gives out 3 plots corresponding to the evolution of the coordinates of the robot and a 3D points cloud animation representing the estimation of the position of the robot and the landmarks and their real values (in order to debug the code, as these values are virtual).
  As we said in the report, an errors occurs in this code because of the covariance matrix of the landmarks not being positive definite. At the end of our project, we were trying to solve this, notably with the next code.

- test_mapping_particle_filter.py : This code is our attempt to replace the mapping using an Unscented Kalman Filter by a mapping using a particle filter.
  It gives out 6 plots, 3 representing the evolution of the estimated coordinates of the drone over the time, and the 3 others representing the evolution of the true position of the first landmark in comparison to its estimation, the true 2D map of the features and their estimation and the evolution of the map estimation error.

5.3. The expected results

A screenshot of the result of the classification and localization of the external features using YOLO is shown in Figure 17, and a screenshot of the modification of YOLO with the labelling algorithm is shown in Figure 16.

Figure 17. Classification and localization of the external features using YOLO

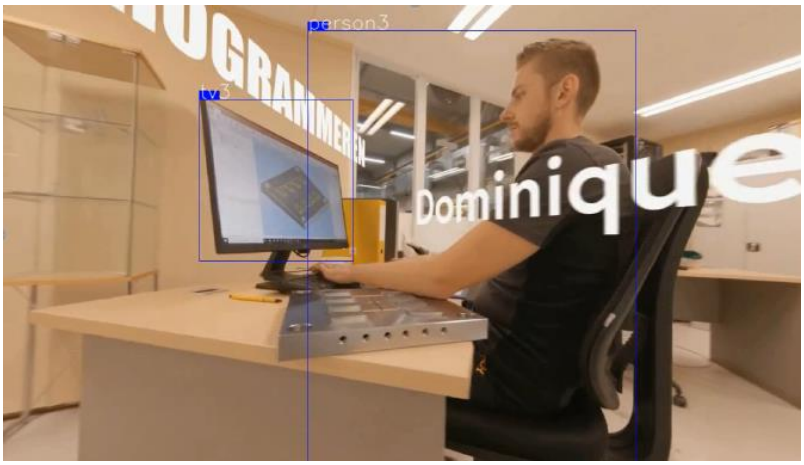


Figure 18. Result of modification of YOLO with the labelling algorithm

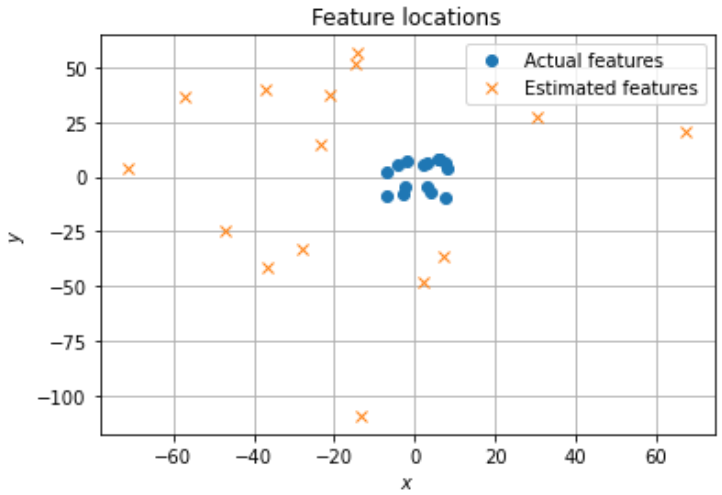


Figure 19 : Output of the SLAM algorithm. The landmarks are represented in 2D for more clarity.

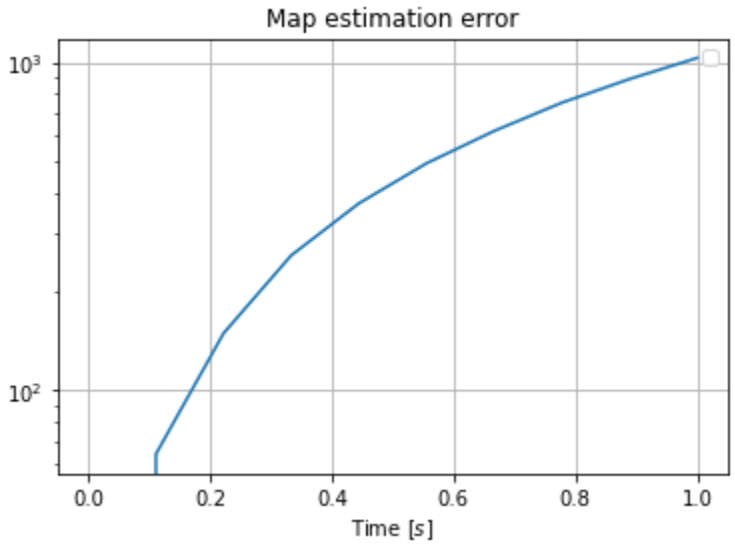


Figure 20. Cumulated error on the estimated position of the landmarks over the time

6. Conclusions

In this project, we started by conducting a study on the various Machine Learning techniques in machine vision, which helped us to create a more general picture of how Machine learning methods can be used to identify and locate external features in the environment.

In order to meet specific conditions such as a precise detection and an accurate classification in order to track well the objects and their position so we needed to use an already existing and efficient algorithm. The most appropriate choice was YOLOv5 which is known as one of the best real time algorithms, it offers a very good accuracy of detection.

To estimate the localization of our drone, the most appropriate solution that we used is to implement a SLAM algorithm (Self Localization and Mapping) so the drone can localize itself anywhere. This allows us to directly use the YOLO algorithm without re-training it as it was trained with the COCO dataset, which can recognize 80 different everyday objects. But this means that we have to create an algorithm that would differentiate two objects of the same category and track them (ex : this is car n°1, this is car n°2).We will also have to determine their position at the same time as that of the robot, and this is why we will need the SLAM algorithm.

References

[1] R. Szeliski, "Computer Vision: Algorithms and Applications.," *Springer*, 2010.

[2] M. &. L. V. &. S. C. &. F. P. Calonder, "BRIEF: Binary Robust Independent Elementary Features," in *Eur. Conf. Comput. Vis.. 6314. 778-792. 10.1007/978-3-642-15561-1_56.*, 2010.

[3] E. Rublee, V. Rabaud and . K. Konolige, "ORB: an efficient alternative to SIFT or SURF.," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2564-2571., 2011.

[4] H. Zhang, "The Optimality of Naive Bayes".*FLAIRS2004 conference.*

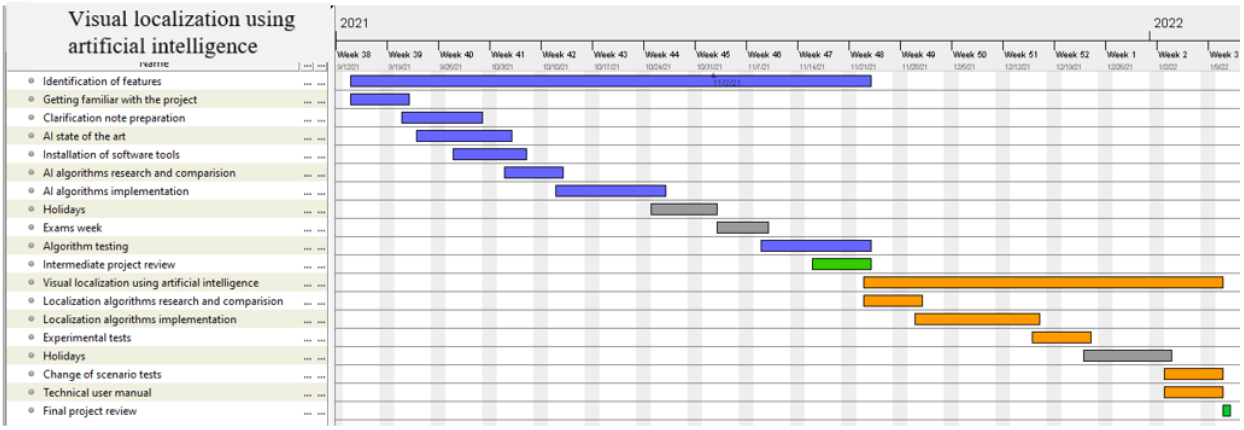
[5] R. B. Girshick, "Fast R-CNN," *CoRR,abs/1504.08083,* 2015.

[6] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once : Inified, Real-Time Object Detection".

[7] [Online]. Available: https://wandb.ai/onlineinference/YOLO/reports/YOLOv5-Object-Detection-on-Windows-Step-By-Step-Tutorial---VmlldzoxMDQwNzk4#installing-cuda.

A. Gant

Gant chart on planning stage



Gant chart in the end