

**École polytechnique de Louvain**

# **Automated tracking of insects**

Authors: **Vahid BEYRAGHI, Jonathan THIBAUT**  
Supervisors: **Olivier BONAVENTURE, Renaud DETRY**  
Reader: **Victor JOOS DE TER BEERST**  
Academic year 2020-2021  
Master [120] in Computer Science and Engineering

## Abstract

Insect identification is a complex task that is mainly executed by professional entomologists that manually distinguish their different characteristics mainly by the use of an identification tree. Our work aims to facilitate this exercise by automating it with the use of neural networks. The goal of this thesis is to provide an automated insect detection and classification trap system that focuses on ten different *Hymenoptera* species such as *Apis*, *Bombus*, *Anthophora* and *Vespula* species.

This master's thesis studies the different solutions in the literature and compares their advantages and disadvantages with respect to the constraints of the project, namely the Faster R-CNN and Mask R-CNN architectures. We inspected similar works in the scope of insect identification to guide our conception choices. We decided to use the Faster R-CNN architecture, as it best matched our constraints, to train a 10 classes model, and used several image augmentation techniques that increased the model's quality during the validation phase. Finally, we deployed our model on an embedded insect trap system to produce detections.

Our model reached an AP score of 73.9% and a ROC AUC value of 71.9% when evaluated on generic data, proving that satisfactory results can be produced with this solution. The precision and recall values of the final configuration that aims to maximize the F1-score reached the values of 76.8% and 74.1% respectively.

The inference on experimental data from the trap resulted in a mean precision and recall for each insect of 57.3% and 53.7% respectively, but that data was really limited as the data harvesting phase was restricted by time and weather constraints. We could however show that it is possible to build a heuristic based on our model's output that could theoretically reach higher values of precision and recall using the hypothesis that the mean precision for each insect is higher than 50% and that consecutive images often correspond to the same insect.

# Acknowledgments

At first, we would like to thank our two supervisors, Pr Olivier Bonaventure and Pr Renaud Detry. We are very grateful to both of them for the time they invested in our project and their continued and frequent support during this master's thesis year. We are also grateful for their precious feedback that led us to our final solution despite the small or important challenges that we faced.

We also would like to thank Mr Robert Lequeux, for his beekeeper's advices as well as Mr Alain Paquet from the Natagora team and Mr Alain Gaillat, for their help and interest in our thesis.

Finally, we would also like to thank our family and friends for supporting us during the whole project.

# Contents

|           |                                                  |           |
|-----------|--------------------------------------------------|-----------|
| <b>1</b>  | <b>Introduction</b>                              | <b>1</b>  |
| <b>I</b>  | <b>Background</b>                                | <b>4</b>  |
| <b>2</b>  | <b>Basic concepts of Machine Learning</b>        | <b>5</b>  |
| 2.1       | Convolutional neural networks . . . . .          | 5         |
| 2.1.1     | Convolutional neural network structure . . . . . | 5         |
| 2.1.2     | Training neural networks . . . . .               | 8         |
| 2.1.3     | Feature extractors . . . . .                     | 10        |
| 2.2       | Evaluation metrics . . . . .                     | 11        |
| 2.2.1     | Accuracy . . . . .                               | 11        |
| 2.2.2     | F-score . . . . .                                | 12        |
| 2.2.3     | ROC AUC . . . . .                                | 13        |
| 2.2.4     | Average Precision . . . . .                      | 14        |
| 2.3       | Machine Learning libraries . . . . .             | 15        |
| 2.3.1     | TensorFlow . . . . .                             | 15        |
| 2.3.2     | Keras . . . . .                                  | 15        |
| 2.3.3     | OpenCV . . . . .                                 | 15        |
| 2.3.4     | Scikit-learn . . . . .                           | 16        |
| 2.4       | Conclusion . . . . .                             | 16        |
| <b>II</b> | <b>Related works</b>                             | <b>17</b> |
| <b>3</b>  | <b>Object detection network architectures</b>    | <b>18</b> |
| 3.1       | R-CNN . . . . .                                  | 18        |
| 3.2       | Fast R-CNN . . . . .                             | 21        |
| 3.3       | Faster R-CNN . . . . .                           | 22        |
| 3.4       | Mask R-CNN . . . . .                             | 26        |
| 3.5       | Conclusion . . . . .                             | 28        |

|            |                                                             |           |
|------------|-------------------------------------------------------------|-----------|
| <b>4</b>   | <b>Insects applications</b>                                 | <b>29</b> |
| 4.1        | Insects traps . . . . .                                     | 29        |
| 4.2        | Insects detection . . . . .                                 | 30        |
| 4.3        | Conclusion . . . . .                                        | 31        |
| <b>III</b> | <b>Automated tracking of insects</b>                        | <b>32</b> |
| <b>5</b>   | <b>Model training</b>                                       | <b>33</b> |
| 5.1        | Goals and constraints . . . . .                             | 33        |
| 5.2        | Methodology . . . . .                                       | 34        |
| 5.2.1      | Architectural choices . . . . .                             | 35        |
| 5.2.2      | Implementation . . . . .                                    | 35        |
| 5.2.3      | Model limitations . . . . .                                 | 40        |
| 5.3        | Experiments . . . . .                                       | 41        |
| 5.3.1      | Initial training . . . . .                                  | 41        |
| 5.3.2      | Validation . . . . .                                        | 42        |
| 5.4        | Results . . . . .                                           | 48        |
| 5.4.1      | Evaluation of the model . . . . .                           | 48        |
| 5.4.2      | Evolution of the metrics for different thresholds . . . . . | 53        |
| 5.4.3      | Examples . . . . .                                          | 55        |
| 5.5        | Conclusion . . . . .                                        | 58        |
| <b>6</b>   | <b>Real-time deployment</b>                                 | <b>59</b> |
| 6.1        | Goals and constraints . . . . .                             | 59        |
| 6.2        | Autonomous trap . . . . .                                   | 60        |
| 6.2.1      | Trap layout . . . . .                                       | 60        |
| 6.2.2      | Motion detection based algorithm . . . . .                  | 63        |
| 6.2.3      | Inference time analysis . . . . .                           | 65        |
| 6.2.4      | Multiprocessing for performance . . . . .                   | 66        |
| 6.2.5      | Cropping to improve quality . . . . .                       | 67        |
| 6.3        | Inference results . . . . .                                 | 69        |
| 6.3.1      | Qualitative analysis . . . . .                              | 73        |
| 6.4        | Conclusion . . . . .                                        | 76        |
| <b>7</b>   | <b>Conclusion</b>                                           | <b>77</b> |
| 7.1        | Further work . . . . .                                      | 79        |
|            | <b>Appendices</b>                                           | <b>87</b> |

|          |                                                       |           |
|----------|-------------------------------------------------------|-----------|
| <b>A</b> | <b>Basic concepts of entomology</b>                   | <b>88</b> |
| A.1      | Insect identification . . . . .                       | 88        |
| A.1.1    | Distinguishing characteristics of an insect . . . . . | 88        |
| A.1.2    | Insects classification . . . . .                      | 89        |
| A.2      | Hymenoptera . . . . .                                 | 90        |
| A.3      | Conclusion . . . . .                                  | 93        |
| <b>B</b> | <b>User manual</b>                                    | <b>96</b> |
| B.1      | The Bees Dataset and Models repository . . . . .      | 96        |
| B.2      | The Faster RCNN repository . . . . .                  | 97        |
| B.2.1    | Project layout . . . . .                              | 97        |
| B.2.2    | Power BI reports . . . . .                            | 100       |
| B.2.3    | Code organization . . . . .                           | 100       |
| B.2.4    | Running scripts . . . . .                             | 103       |

# List of abbreviations

**AP** : Average Precision

**API** : Application Programming Interface

**AUC** : Area Under Curve

**CNN** : Convolutional Neural Network

**CPU** : Central Processing Unit

**DC** : Direct Current

**DNA** : DeoxyriboNucleic Acid

**FCN** : Fully Convolutional Network

**FP** : False Positive

**FPR** : False Positive Rate

**FN** : False Negative

**GPU** : Graphical Processing Unit

**IoU** : Intersection over Union

**mAP** : mean Average Precision

**MLP** : MultiLayer Perceptron

**RAM** : Random Access Memory

**R-CNN** : Region-based Convolutional Neural Network

**ReLU** : Rectified Linear Unit

**ResNet** : Residual Network

**RGB** : Red Green Blue

**ROC** : Receiver Operating Characteristic

**RoI** : Region of Interest

**RPN** : Region Proposal Network

**SGD** : Stochastic Gradient Descent

**SVM** : Support Vector Machine

**TN** : True Negative

**TP** : True Positive

**TPR** : True Positive Rate

**VGG** : Visual Geometry Group

**VIA** : VGG Image Annotator

# Chapter 1

## Introduction

Insects are very beneficial to humans in different ways. Some can be beneficial to reduce the population of pests that are threatening crops humans rely on for food. They also have a major role in pollination. Indeed over 75% of all flowering plants and 75% of the crops rely on insects for reproduction, to carry pollen from one flower to another. Finally, lots of insects are also good recyclers as they feed on dead plants and animals and break them down into nutrients that help plants grow [1]. It is therefore important to count and identify insects in order to have a spatial distribution of them. In this thesis, we focus on a range of *Hymenoptera* species endemic to European countries. This is one of the most beneficial insect order to humans [2].

There are two main categories of methods to count and recognize insects. The first one needs human intervention. It consists in designing traps that capture insects. Then specialists identify the insects via well defined methods like the use of determination key by observing their characteristics. The second category consists in automating the identification. This is done by using object detection techniques in the field of Machine Learning. With this method, there are again several possible approaches. One of them also involves the creation of a lethal trap that captures insects. Some pictures of the captured insects are then taken and machine learning algorithms are used to identify the species. This approach partially involves human assistance. Another approach consists in taking pictures of insects in real-time. When an insect is detected, a picture of the insect is taken and is then processed by a machine learning algorithm able to identify the detected species.

Today and for the last 15 years, machine learning has become more and more popular and is in complete expansion due to the increasing performance of computer calculations. This computer science field can be applied to a very large range of topics and involves many different techniques. One of them is the use of Region-based Convolutional Neural Networks (R-CNNs) in the field of object detection and identification on an image. These networks are designed to identify on images

the regions that correspond to the classes of interests [3, 4].

The goal of this thesis is to automate the tracking and the identification of ten different insect species so that a human intervention is no longer needed for such a task. This identification is done by taking pictures of insects in real-time through a camera trap at first and by running a machine learning algorithm then. The objective of this thesis is to deploy an application that performs such a task on a Raspberry Pi. There already exists several solutions to automatically identify insects with machine learning but they often focus on a small range of species and most of them are deployed in Asia [5, 6, 7]. Our solution differ from them as we focus on European species and on more insect classes. Globally, this thesis faces two main challenges.

The first challenge is the training of a model built with a well chosen object detection network architecture. This architecture needs to present high performance and should fit the constraints of the physical application. Indeed, the model should be complex enough to distinguish ten different species but should also be able to run on the physical prototype. Indeed, the hardware that is used for the trap is a Raspberry Pi 4 B. Such a board is built up with a quad core 64 bits Central Processing Unit (CPU) based on a clock speed of  $1.5\text{Ghz}$ . This CPU and the lack of a powerful GPU involve performance constraints [8].

The second challenge consists in designing a physical trap able to attract the species of interest in front of a camera lens. The embedded system should be able to use our model on high quality pictures taken automatically, and in real-time, by the camera, to identify the insects present in front of it. Moreover, this trap must be designed to be completely autonomous and weatherproof.

To face the mentioned challenges and achieve our goals, we first proceeded in an iterative way to train our model. At first, we chose the network architecture that corresponds the most to our problem and defined some choices of conception. The chosen architecture is Faster R-CNN. Though it was published in 2015, it still performs well nowadays in the field of object detection [9]. Then, we trained the algorithm with an iterative procedure. At first, we conducted several experiments on a small subset of classes to validate the choice of our model parameters. After this validation phase, we trained our model on a dataset that contains all the species of interests for the purpose of deploying it on the final trap system. In parallel of the model deployment, we conducted some experiments on the trap to determine the best strategy to attract insects in front of the camera lens and take high quality pictures of them. We built the physical trap iteratively since we modified its design according to the results obtained during the experimental phase.

In the long run, the modeled trap could be deployed in any region of Belgium to automatically track insects. The data can namely be used for insect census.

Moreover, with such a trap, we can also get statistical data such as weather information collected at the insect detection time.

## Structure of the master's thesis

This master's thesis is composed of three main parts. Each part starts with a short introduction that gives a general overview of its content and then develops its related topics in more details.

The part I contains the work's background. This part explains the most important machine learning concepts on which the presented work is based, namely the global functioning of a Convolutional Neural Network (CNN) and the evaluation metrics that are used to assess the model.

The part II considers other works related to the one presented in this thesis. It discusses how the presented work differs from the existing ones. This part is separated in two chapters. The chapter 3 presents the object detection network architectures and their respective performance nowadays. In this chapter, we analyze the characteristics of the different architectures considered, in order to justify our choice. The chapter 4 presents some similar works that use machine learning to identify insects through object detection techniques.

The part III of the report focuses on the concrete application, the automated tracking of insects. It contains two different chapters. The chapter 5 deals with the training of the model. It describes the whole methodology we used to build our model and explains the different experiments based on it. Then, it evaluates the quality of that model along with its inference results. The chapter 6 presents the physical autonomous trap and the final application of the work. We describe how we imagined and conceived the trap to attract the insects, deployed our model in a real-time application and present its results.

Finally this thesis ends up with a general conclusion presented in chapter 7 that also opens up some prospects of improvement, followed by some appendices containing useful information to get a better and more detailed comprehension of the whole project.

# Part I

## Background

### **Contents**

This part contains one chapter which aims at providing the necessary explanations of machine learning concepts used as cornerstone of this project. More specifically, it describes the structure of convolutional neural networks and how they are trained. It also mentions some evaluation metrics that we used in our final model assessment. Finally, it also briefly reviews some important Python libraries in the field of machine learning.

## Chapter 2

# Basic concepts of Machine Learning

In this chapter, we describe some key concepts in machine learning on which the solution presented in part III rests.

At first we focus on the CNN's functioning. We describe how the different CNN's layers work, namely the convolutional layers, the pooling layers and the multilayer perceptrons (MLP). We also explain the training phase of such a network and present some popular feature extractors.

Secondly, we focus on some popular metrics we can use to evaluate a machine learning model. These are useful to analyze our application results.

Finally, the chapter also contains a short description of some Python libraries useful in machine learning.

## 2.1 Convolutional neural networks

Convolutional neural networks are probably one of the most popular machine learning technologies. They are used in a variety of applications, including image recognition.

In short, image recognition consists in identifying objects, people, places, action, texts, on a given image by analyzing its pixel values. This machine learning process involves different phases and components of the CNN structure. These are described in this section.

### 2.1.1 Convolutional neural network structure

Typical CNN architectures for image classification consist in different building blocks. The feature extraction phase is done through multiple convolutional and

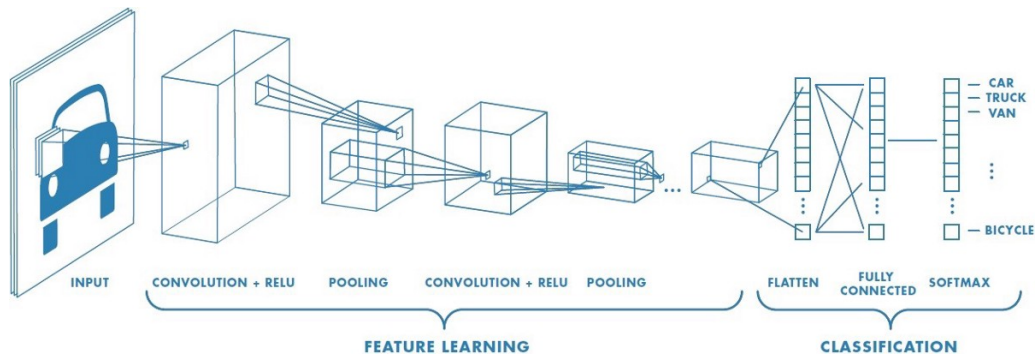


Figure 2.1: Typical CNN architecture [10].

pooling layers. Then a multilayer perceptron performs the classification of the image. It is composed of fully connected layers and a softmax layer. This is illustrated on figure 2.1.

### Convolutional layers

The convolutional layer's goal is to extract the main features of the input image (and potentially to reduce its dimensionality) and to turn it into a feature map. This is done by applying a filter on all possible regions of the image, as represented on figure 2.3 (a). As one can see on figure 2.2 (a), with this method, each neuron of the convolutional layer only contains information about a restricted number of input neurons. The green neurons correspond to the image input neurons while the blue ones correspond to the features extracted by convolution. Each coloured edge between neurons corresponds to a fixed weight. Thus, for example, in the presented figure, the convolution phase only involves three different weights represented by red, green and blue edges. These weights values are successively slid along the different possible neuron windows, that correspond to the possible regions on which the convolutional mask is applied. In fact, a lot of filters are being applied during the process and the goal of the whole CNN training is to select the interesting filters.

Multiple parameters can be tweaked in this layer, such as the size of the filter, its shape and the padding (i.e. the stride length when applying the filter). The latter allows us to control the reduction of the input dimensionality through the layer as having a unit stride length does not reduce the dimensionality while higher values reduce it in a linear manner. It is not necessary to limit the number of convolutional layers to one. In a multiple convolutional layers network, the first layer essentially maps the low-level features of the input image, while the following layers increasingly notice higher-level features, up to similar features than the ones a human would have in his understanding of the input image.

## Pooling layers

The use of pooling layers is mostly to learn a hierarchy of spatial abstractions and to verify a property of translation invariance: even if the image is translated, it is guaranteed that its features are still detected and the expression output by the pooling phase is maintained as consistent as possible. One method is to divide the input in sections and return for each of these sections, the maximum value of their elements. This is called max pooling. This process is illustrated on figure 2.3 (b). It is useful to extract main features that are positional and rotational invariant, while having a noise suppression effect (because we only consider the maximum values). Another method is the average pooling method, which consists in returning the average of the elements of each section [11, 10].

## Multilayer perceptron

The MLP consists of at least three layers of nodes.

Once all the previous steps are done, it is necessary to make data suitable for the multilayer perceptron that aims at classifying the initial input image. This is the role of the flatten layer. This layer is the input layer of the perceptron and corresponds to the suitable image reshaped as a column vector.

The next layers, called hidden layers are fully connected. In opposition to the convolutional layers, they exploit every data neuron of the input layer, as shown on figure 2.2 (b).

In the hidden layers, each neuron has an activation function that produces an intermediate output based on intermediate weighted inputs. The weights do not have a fixed value. The weights values are learned by backward propagation of errors (backpropagation). This is an example of supervised learning algorithm. An error function is defined to compare the expected output of the model and its actual output. The method calculates the gradient of the error function with respect to the neural network's weights. The 'backwards' term stands for the fact that the gradient's calculation proceeds backwards through the network, with the gradient corresponding to the final layer being calculated first and the gradients corresponding to the first layer being calculated last. This process can be done using the gradient descent algorithm [12], explained in more details in section 2.1.2.

The last hidden layer is built using the popular softmax activation function. The goal of this function is to produce the probability distributions of a list of potential outcomes.

The final layer is the output layer. This one corresponds to the final value(s) of the previous computation that allow(s) the algorithm to classify the inputs. This output layer uses the probability distributions returned by the softmax activation function to determine the most likely class corresponding to the initial input picture.

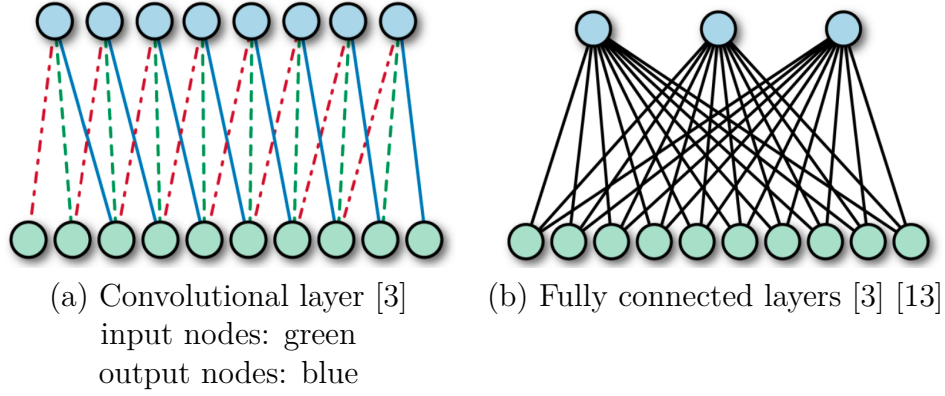


Figure 2.2: Convolutional and fully connected layers.

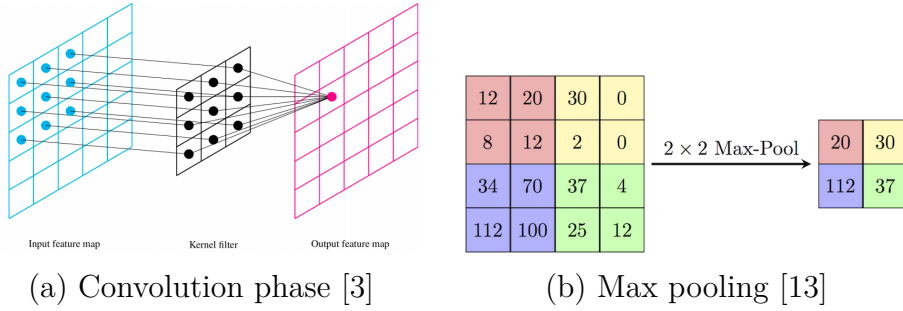


Figure 2.3: Convolutional and max pooling phases.

### 2.1.2 Training neural networks

Once the structure of the network is fixed, the networks' weights should be updated to produce an output as efficient as possible. This phase involves supervised learning. The algorithm compares the expected output of the network with its actual output via a loss metric. The loss value is then used to update the weights' values through a stochastic gradient descent phase.

There are several classical loss functions used to evaluate a CNN. A typical one used in multi-class classification problems is the categorical cross-entropy loss. This metric is defined in equation 2.1.

$$CE(f(s), t) = - \sum_i^C t_i \log(f(s)_i) \quad (2.1)$$

$$f(s)_i = \frac{e^{s_i}}{\sum_j^c e^{s_j}} \quad (2.2)$$

The expression  $f(s)_i$  defined in equation 2.2 corresponds to the output of the activation function performed on the softmax layer.  $C$  corresponds to the number of classes considered by the model.  $t$  corresponds to the target vector, which is one-hot encoded. That means that the only non-zero component corresponds to the expected output class [14].

Once the loss function is calculated at the very end of the network, the stochastic gradient phase aims at updating weights' values following formula 2.3

$$w = w - \eta \cdot \nabla_w CE(f(s), t) \quad (2.3)$$

where  $w$  represents the network's weights and  $\eta$  corresponds to the learning rate. In practice, this formula can be decomposed according to the different networks layers by using the chain rule. The stochastic gradient descent phase happens every time a new input is presented to the network and until convergence of the loss value towards a minimum.

Often, an optimizer is used to update the stochastic gradient descent phase during the training phase. A popular one in CNN computation is the Adam optimizer [15]. This computes adaptive learning rates for each parameter. To simplify the notations, let  $g_t$  denote the gradient vector computed at the  $t^{th}$  stochastic gradient descent (SGD) phase. The Adam optimizer computes the decaying averages of past and past squared gradients  $m_t$  and  $v_t$  via the formulas presented on equations 2.4 and 2.5 in which  $\beta_1$  and  $\beta_2$  are optimizer's parameters that could eventually evolve through SGD steps.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (2.4)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (2.5)$$

As  $m_t$  and  $v_t$  are initialized to 0, they are biased to 0 across SGD phases. To counteract these biases, some expectations of these variables should be computed with formulas 2.6 and 2.7

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (2.6)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (2.7)$$

Those updated parameters are then concretely used by the optimizer in the Adam update rule written on equation 2.8 [15].

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (2.8)$$

### 2.1.3 Feature extractors

This section aims at presenting two popular backbones which correspond to the features extractor networks that are theoretically described in section 2.1.1. In practice, when training a CNN with new classes, it is often useful to perform transfer learning. This consists in reusing a pre-trained model as initial model. This initial model determines the initial weights that are updated during SGD phases and the architecture of the extractor.

A popular pretrained CNN model is VGG16 which means Visual Geometry Group with 16 layers [16]. This model was trained for weeks using a dataset called ImageNet. This dataset contains over 15 millions labeled high-resolution images belonging to roughly 22,000 categories. The network inputs correspond to down-sampled images to a fixed resolution of 256x256. Its architecture is presented on figure 2.4.

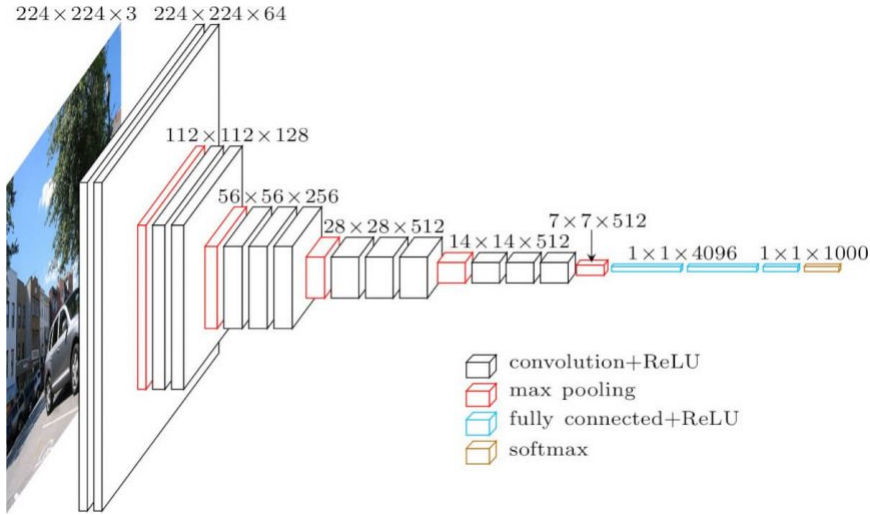


Figure 2.4: VGG16 Architecture [16].

The input images pass through different convolutional layers with a 3x3 receptive field. The convolution stride is fixed to 1 so that the spatial resolution is preserved after convolution. This architecture presents five max-pooling phases performed over a 2x2 window with stride 2. The MLP that follows the convolutional layers involves three fully connected layers before the softmax layer. The first two fully connected layers have 4096 channels each and the third one contains 1000 channels. All hidden layers are equipped with a Rectified Linear Unit (ReLU), which leads to good performance [17]. It corresponds to an activation function that outputs a value equal to the input only if it is positive. The advantage of this activation function is that, compared to classical ones, its output is not bounded. By this

property, this activation function tends to avoid vanishing gradients problems. This problem happens when the output of the activation functions are mapped to values close to 0. This pushes the gradient to vanish by having almost-zero values too [16].

Other interesting network architectures are residual networks (ResNet) [18]. Those architectures have the particularity to be deeper and less wide than classical ones. Increasing the depth of the network instead of widening it results in fewer extra parameters. This would also increase the risk of vanishing gradients problems. To avoid this, some layers of residual networks give information to several next layers and not only one, as represented on the left part of figure 2.5. This network particularity fastens the training of the algorithm.

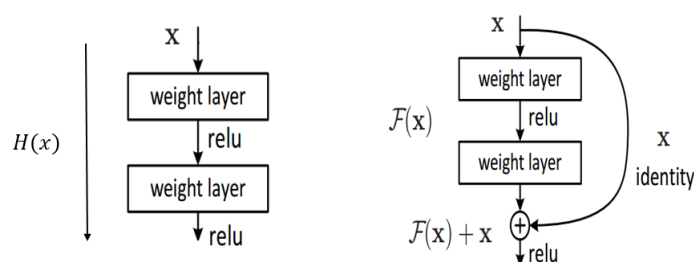


Figure 2.5: Comparison between classical architecture and ResNet architecture [18].

## 2.2 Evaluation metrics

In machine learning, once the model has been trained, the next step consists in evaluating its performance. To do so, there exists several metrics that are more or less relevant according to the final goal of the model.

### 2.2.1 Accuracy

Let us consider a machine learning model that has to predict if an instance is true or false. If we compare the prediction of the algorithm with the actual label of the instance, the prediction can either be:

- True Positive (TP): The actual and predicted labels are both true
- False Positive (FP): The actual label is false but the predicted one is true.
- False Negative (FN): The actual label is true but the predicted one is false.

- True Negative (TN): The actual and predicted labels are both false

These concepts can be summarized in a confusion matrix, presented in table 2.6.

| Predicted \ Actual class | Positive | Negative |
|--------------------------|----------|----------|
|                          | TP       | FP       |
| Positive                 | TP       | FP       |
| Negative                 | FN       | TN       |

Figure 2.6: Confusion matrix.

With this matrix we can then define an accuracy metric which corresponds to the ratio of the number of correct predictions on the total number of predictions. The definition of this metric is written in equation 2.9 [19].

$$ACC = \frac{TP + TN}{TP + FP + FN + TN} \quad (2.9)$$

### 2.2.2 F-score

However, the accuracy could not be a good metric for the evaluation of the model. For instance, if we evaluate the model on a imbalanced dataset, building a model that always predicts the class that is more present than the others leads to a good accuracy. To correctly evaluate a model when this scenario happens, one can also define two more useful metrics based on the confusion matrix, which are the precision and the recall.

The precision corresponds to the ratio of the number of true labels correctly predicted on the total number of true labels predicted. This metrics is defined in equation 2.10.

$$Prec = \frac{TP}{TP + FP} \quad (2.10)$$

The recall corresponds to the fraction of samples from a class which are correctly predicted by the model. This metric is defined in equation 2.11

$$Rec = \frac{TP}{TP + FN} \quad (2.11)$$

Depending on the application, it can be more interesting to give a higher priority to recall or precision. However, in some situations, both metrics are important. In such a case, one can evaluate the model with the *F1 – score*, defined in equation 2.12 [19].

$$F1 - score = 2 * \frac{Prec * Rec}{Prec + Rec} \quad (2.12)$$

### 2.2.3 ROC AUC

In some applications, the model predicts a label with a certain probability. With such a model, one can observe its performance when the cut-off threshold varies. The Receiver Operating Characteristic (ROC) curve shows the true positive rate (TPR) against the false positive rate (FPR) for various threshold values. The TPR corresponds to the recall metric defined in equation 2.11 and the FPR is expressed in equation 2.13.

$$FPR = \frac{FP}{FP + TN} \quad (2.13)$$

By decreasing the prediction threshold, both FPR and TPR increase as if this threshold is low, the model more easily predicts positive classes. But it also means that the model can more easily predict classes as positive classes that are actually negative. The ROC curve is drawn in blue in figure 2.7.

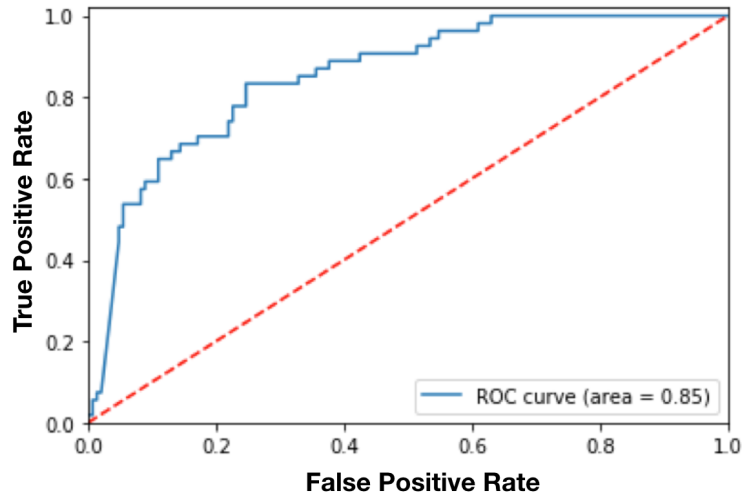


Figure 2.7: ROC AUC [19].

One can then consider that a model is good if for a low FPR, it presents a high TPR, which means that it more often correctly classifies true labels than it badly classifies true labels. This means that the more convex the ROC curve is, the better the model is. Thus, one can calculate the Area Under the ROC Curve (ROC AUC). A model that presents a good ROC AUC value can then be considered as a good model [19].

## 2.2.4 Average Precision

When building a model that involves a decision threshold of probability, one can also define the Average Precision (AP) metric, which corresponds to the area under the curve of the precision of the model varying according to the recall of the model.

If the decision threshold decreases, the model more easily predicts some instances as positive labels, which means that the recall increases. In other words, among all the actual positive classes, more of them are predicted as positive. But decreasing this decision threshold also means that some actual negative classes can be predicted as positive classes. So, the precision value decreases. Hence, one can consider a good model as a model that presents a good recall value, even if its precision value is important. This means that the more convex the precision-recall curve is, the better the model is. This convexity of the curve can be quantified with the area under the precision curve. Thus, a good model is a model that presents a large area under its precision-recall curve (AP). The precision-recall curve of a model that performs well is presented in figure 2.8 [19].

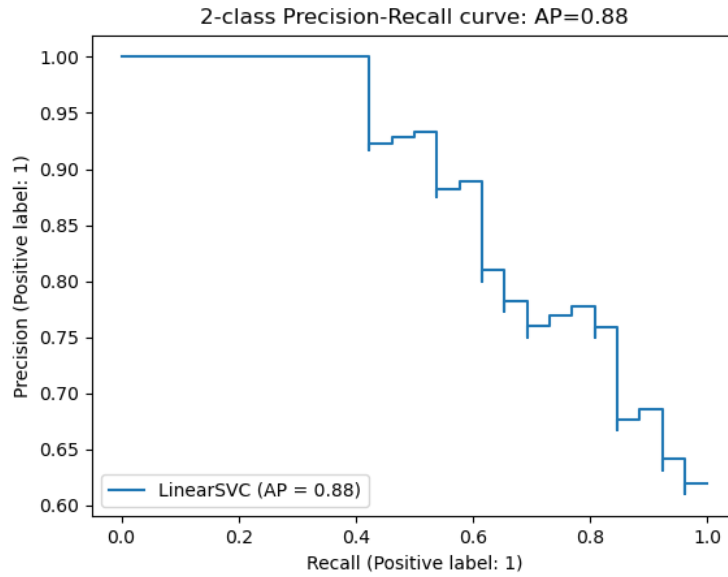


Figure 2.8: Precision-recall curve [20].

## 2.3 Machine Learning libraries

Python language proposes a range of useful libraries in the machine learning field. Some of them are very useful in CNN training for image classification tasks. The following sections briefly describe the main Python libraries used in this work.

### 2.3.1 TensorFlow

TensorFlow is an open-source software library dedicated to machine learning algorithms. With this technology, developers can easily work on applications by using community resources. Hence, it is possible to use a pre-trained model of the library and adding new classes (as done in this article [21]). With such a library, it is also possible to easily develop and train a complete convolutional neural network.

Tensorflow offers a module named *compat.v1* which provides a set of compatibility functions between Tensorflow versions 1.x and 2.x. In practice, this module can be useful to create an interactive session. An interactive session is an object in which *Operation* objects are executed and *Tensor* objects are evaluated. In practice, such a session can be initialized with a *ConfigProto* object that allows the user to choose the device on which the algorithm is running (Graphical Processing Unit (GPU) or CPU) by using its *gpu\_options* attribute.

### 2.3.2 Keras

Keras is a central part of TensorFlow that gives the opportunity to create model and configure them with meta-parameters. *Keras.models* module contains *Model* objects that can be configured with several layer objects, contained in *Keras.layers*. Among them, one could namely mention the *Input* layer used to instantiate a Keras tensor, the *Conv2D* layer that creates a convolution kernel that is convolved with the input layer to produce a tensor of outputs, the *MaxPooling2D* layer that downsamples the input layer by taking the maximum value over a window defined by the *pool\_size* attribute. These latter ones can be configured with several parameters like the stride value, mentioned in subsection 2.1.1.

### 2.3.3 OpenCV

OpenCV [22] is a library that offers many useful methods for image processing techniques. One of the most important feature offered by this library is the resizing of images. Indeed, as mentioned in section 2.1.3, features extractors are configured with a fixed input image size. It is also useful to augment the image dataset by flipping images with the *flip* method. Finally, the library can be used in the image

preprocessing phase to apply box filtering techniques or Gaussian blur techniques on the images. These features are explained in more details in subsection 5.3.2.

### **2.3.4 Scikit-learn**

Scikit-learn [23] is a Python library with which it is possible to implement classification, regression or clustering machine learning algorithms. It also proposes a large choice of metrics to evaluate the quality of a model's predictions. Among the metrics functions, one can namely mention the accuracy score, the AP score or the area under the Receiver operating characteristic curve. These evaluation metrics are described in section 2.2.

## **2.4 Conclusion**

To conclude, the image recognition techniques and object detection techniques rely on convolutional neural networks, which are made up of convolution layers and pooling layers. The former ones aim to extract the main features of the input images while the later ones aim to assure the translation invariance property. Once a model is built, one should evaluate it with some specific chosen metrics, depending on the objective of the model's application. Among these metrics, one can namely cite the average precision and the area under the receiver operating characteristic curve that we will reuse in the rest of this report.

# Part II

## Related works

### Contents

This second part presents diverse works more or less closely related to our master's thesis. It is composed of a chapter that presents some popular CNN architectures used in the object detection field. We gradually present these architectures from the least elaborated one to the most complex ones. They are compared to each other, with their respective benefits and drawbacks, in order to define the candidate architectures for our projects. The next chapter aims at presenting different trap designs. It also considers several different insect detection systems and their respective performances evaluation. This gives a guideline for the performances we can expect our solution to reach.

# Chapter 3

## Object detection network architectures

CNNs as described in section 2.1 have shown really good results for image recognition. However, there has been a lot of progress in other applications such as object detection and semantic segmentation. Multiple architectures based on CNNs to solve these problems have been proposed and iteratively improved in the literature. Some of these solutions are described in this chapter, along with their advantages and disadvantages.

### 3.1 R-CNN

Whereas CNNs work well for image recognition, they are not adapted to tackle the problem of object detection. Indeed, this problem consists in detecting instances of semantic objects of a certain class in images. The issue with CNN is that the length of the output layer would be variable as the number of instances of the object in the image is not fixed.

A first correct approach for solving this issue was R-CNN. It uses a selective search to extract 2000 regions from the images (called region proposals). This method allows to work with a limited number of regions on the images compared to the naive approach that would be to select a huge amount of (random) regions from the initial image to account for the possible different aspects ratios and locations of the instances within the image, as well as for the unknown number of instances on each image. This selective search algorithm is detailed in algorithm 1, though the main idea is to generate many candidate regions on the image and to combine similar regions into larger ones. These larger regions are then used as

final candidate region proposals.

---

**Algorithm 1:** Selective search algorithm [4].

---

**Input** : (colour) image  
**Output** : Set of object location hypotheses  $L$   
Obtain initial regions  $R = \{r_1, \dots, r_n\}$  using Felzenszwalb and Huttenlocher (2004) [24];  
Initialize similarity set  $S = \emptyset$ ;  
**foreach** *Neighbouring region pair*  $(r_i, r_j)$  **do**  
    Calculate similarity  $s(r_i, r_j)$ ;  
     $S = S \cup s(r_i, r_j)$ ;  
**end**  
**while**  $S \neq \emptyset$  **do**  
    Get highest similarity  $s(r_i, r_j) = \max(S)$ ;  
    Merge corresponding regions  $r_t = r_i \cup r_j$ ;  
    Remove similarities regarding  $r_i$  :  $S = S \setminus s(r_i, r_*)$ ;  
    Remove similarities regarding  $r_j$  :  $S = S \setminus s(r_*, r_j)$ ;  
    Calculate similarity set  $S_t$  between  $r_t$  and its neighbours;  
     $S = S \cup S_t$ ;  
     $R = R \cup r_t$ ;  
**end**  
Extract object location boxes  $L$  from all regions in  $R$ ;

---

Each of these regions proposals are fed in a large CNN. The CNNs act as feature extractors for each of these regions as they produce features that can be used by class-specific linear support vector machines (SVMs).

SVMs are supervised machine learning models that analyze data for classification and regression analysis. The main idea behind them is to fit a hyperplane between the clusters of data in the features space, that maximizes the margins between the hyperplane and its closest data points (see figure 3.1).

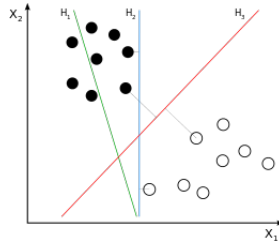


Figure 3.1: 2D representation of two non-optimal hyperplanes ( $H_1$ ,  $H_2$ ) and of the optimal hyperplane ( $H_3$ ) to separate two clusters of data [25].

The output of each SVM for each region indicates whether this region contains

an instance of a specific class. Thus, R-CNN is capable of indicating the regions that correspond to instances of the classes on the initial image. The overview of this architecture is represented on figure 3.2.

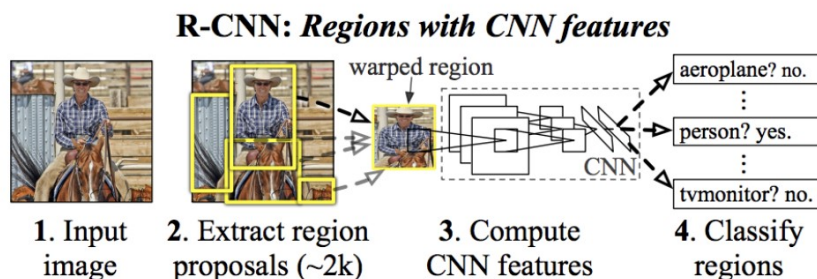


Figure 3.2: R-CNN architecture overview in 4 steps: (1) taking the input image, (2) generating around 2000 region proposals, (3) computing the features of each proposal region using a CNN, and (4) classifying each region using class-specific SVM's [4].

The main issue with this architecture resides in the fact that each of the 2000 region proposals per image must be processed by CNN. This implies that the training and inference times of the models using this architecture are really high (which prevents using this kind of architecture in real-time). Another flaw is that the selective search algorithm is a fixed algorithm, and that means that there is no learning at that stage (the quality of the region proposals does not improve with the training) [4, 26].

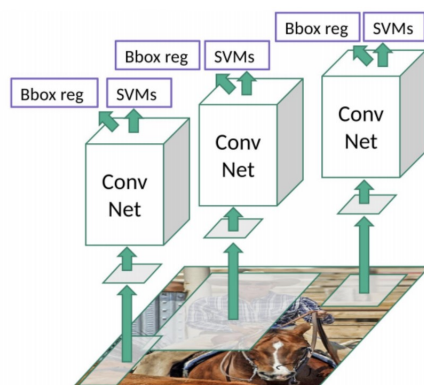


Figure 3.3: Each of the 2000 regions proposed by the selective search algorithm are fed in a CNN which limits the performance [26].

## 3.2 Fast R-CNN

The main drawbacks of R-CNN were partially addressed with Fast R-CNN. Fast R-CNN improved the previous architecture by processing each image only once through the CNN to produce a feature map on which the regions of interest could be identified (using the selective search method detailed in section 3.1). A region of interest (RoI) pooling layer was added at that stage to reshape the region of proposals identified in a fixed size that could be fed into fully connected layers. These layers output a RoI feature vector (for each RoI) that is fed to the two output layers of the system: a softmax layer in order to predict the class of the instance contained in that RoI and a layer that outputs four real numbers encoding the bounding-box position of the instance. See the overview of that architecture on figure 3.4.

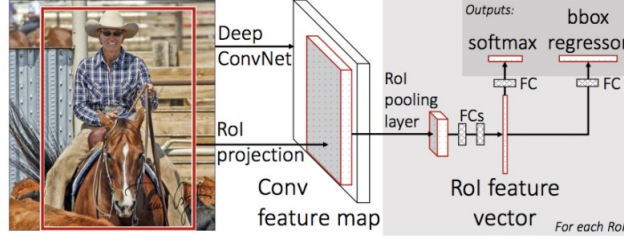


Figure 3.4: Overview of the Fast R-CNN architecture [27].

The training of the network is based on the minimization of the following multi-task<sup>1</sup> loss function:

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

in which each training RoI is labeled with a ground-truth class  $u$  and a ground-truth bounding-box target  $v$ .  $p = (p_0, \dots, p_K)$  is the probability distribution (per RoI) produced by the classification output layer over  $K + 1$  categories, and  $t^k = (t_x^k, t_y^k, t_w^k, t_h^k)$  is the bounding-box regression offsets for each of the  $K$  classes, indexed by  $k$ , produced by the regression output layer. The Iverson bracket indicator function  $[u \geq 1]$  is 1 when  $u \geq 1$  and 0 otherwise. This allows the loss function to ignore the background class (that is labeled  $u = 0$  by default) as there is no notion of ground-truth bounding-boxes for background RoIs. To determine if a predicted bounding-box corresponds to a known class or a background, the algorithm considers all the ground-truth boxes and compares them with the

<sup>1</sup>We use the term multi-task loss because that loss is used to train jointly the classification and the bounding-box regression.

modified predicted boxes to find the best one, by computing the intersection over union (IoU) value between the two boxes, which corresponds to the intersection area between a predicted box and its corresponding ground-truth box. If the best predicted box has an IoU value lower than 10%, it skips the ground-truth box. This threshold appears to act as a heuristic for hard example mining. That means that the algorithm did not predict any accurate box that corresponds to the considered ground-truth box. If this IoU value belongs to domain  $[10\%; 50\%]$ , it considers that the best predicted box corresponds to the background. Otherwise, it predicts that this box corresponds to the most likely class. The  $\lambda$  hyper-parameter controls the balance between the two tasks of the loss function (and equals 1 by default). The first task loss  $L_{cls}$  is the log loss for class  $u$  and the second task loss  $L_{loc}$  is defined over a tuple of true bounding-box regression targets for class  $u$ ,  $v = (v_x, v_y, v_w, v_h)$ , and a predicted tuple  $t^u = (t_x^u, t_y^u, t_w^u, t_h^u)$ , again for class  $u$ . These loss functions are computed as such:

$$L_{cls} = -\log(p_u),$$

$$L_{loc} = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L_1}(t_i^u - v_i),$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}.$$

As mentioned above, the image is only run through the CNN once to generate a unique feature map, while with R-CNN, each image's 2000 region proposals had to go through the CNN. This greatly increases performance for training and testing, although it does not solve all the drawbacks of R-CNN listed in section 3.1 [27].

### 3.3 Faster R-CNN

Faster R-CNN did however fix these issues by replacing the selective search used in both R-CNN and Fast R-CNN by a Region Proposal Network (RPN). The RPN takes the resized input image (so that the width is 600px and the length does not exceed 1000px), which is fed in the backbone<sup>2</sup> to produce a feature map. To allow the network to learn where the objects are present on the input image, a set of anchors are generated. These are possible locations for objects, in different sizes and aspect ratios (see figure 3.5).

---

<sup>2</sup>Backbone: Refers to the feature extractor network used.

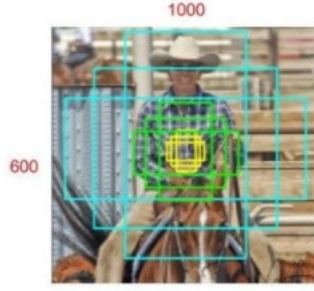


Figure 3.5: Possible anchors in the input images with 3 different sizes ( $128^2$ ,  $256^2$ ,  $512^2$ ) combined with 3 different aspect ratios ( $1 : 1$ ,  $1 : 2$ ,  $2 : 1$ ) [28].

The network tries to evaluate whether these anchors actually contain an object and refine the anchors coordinates (by combining them notably) into RoIs. To achieve this, the feature map provided by the backbone is passed through an  $n \times n$  convolutional layer to output a 512-d (in the case where the backbone used is VGG) feature map for each possible location. This layer is followed by two branches. These are composed of  $1 \times 1$  convolutional layers whose outputs are fed into fully connected layers. One of these layers corresponds to the box-regression layer (reg), which aims to predict the bounding boxes (of the RoIs) by improving the coordinates of the anchors; and the other is the box-classification layer (cls) which produces a probability, for each anchor, of containing an object.

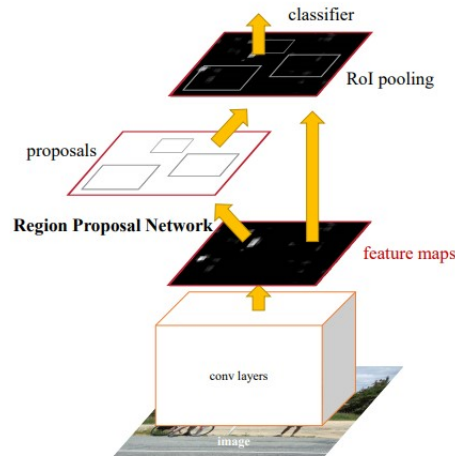


Figure 3.6: Faster R-CNN architecture overview [28].

The backbone used by the RPN is shared with the rest of the Faster R-CNN network such that the feature map produced is also used in a pooling layer, to

extract the features referring to each RoI using the output of the RPN, comparably to what was done in Fast R-CNN with the output of the selective search. The rest of the network is, again, quite similar as what is done in Fast R-CNN in the previous section (section 3.2): the pooling layer is followed by two fully connected layers, which are themselves fed into the sibling classification and regression branches, producing the RoI object classification, as well as the bounding boxes of that RoI (see figure 3.7).

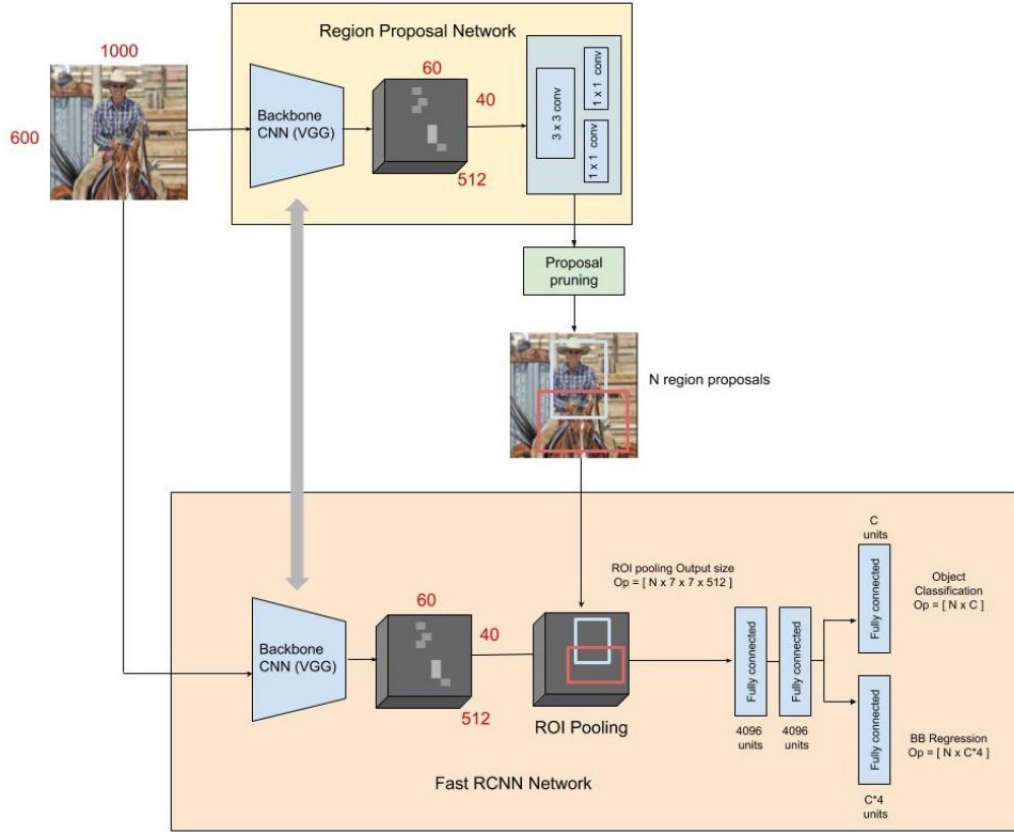


Figure 3.7: Faster R-CNN detailed architecture [28].

The addition of the RPN allows the region proposal to be trained as well. Indeed, the training of the RPN relies on the minimization of the following loss function:

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

For each image,  $i$  is the index of an anchor and  $p_i$  is the predicted probability of anchor  $i$  being an object.  $P_i^*$ , which corresponds to the ground truth label, is 1 when the anchor is positive, and 0 when the anchor is negative. A positive anchor is defined as an anchor that has the highest Intersection-over-Union (IoU) or as an anchor that has an IoU overlap that is higher than 0.7 with any of the ground-truth boxes. Negative anchors are defined as non-positive anchors that have an IoU overlap lower than 0.3 for all ground-truth boxes. Anchors that are neither defined as positive nor negative are not used in the computation of the loss function. The classification loss  $L_{cls}$  is the log loss over the classes *object* and *not object* and the regression loss  $L_{reg}$  is defined over the  $t_i$  vector containing the 4 coordinates of the predicted bounding-box, and  $t_i^*$  vector containing the coordinates of the ground-truth box corresponding to a positive anchor. It is computed as follows:

$$L_{reg}(t_i, t_i^*) = R(t_i - t_i^*),$$

where  $R$  is the robust loss function ( $smooth_{L_1}$ ) defined in the previous section. The two terms of the RPN loss function are normalized by  $N_{cls}$  and  $N_{reg}$  which are respectively the mini-batch size and the number of anchor locations. The  $\lambda$  hyper-parameter carries the same role as described in the previous section.

The training of this architecture can be done in two main ways:

- *Alternating training:* This solution rests on the fact that both the RPN and the classifier (the Fast R-CNN network) can be trained end-to-end by back-propagation and SGD. The idea is to first train the RPN, and use the proposals to then train Fast R-CNN. The network tuned by Fast R-CNN can subsequently be used to initialize RPN and these operations can be repeated iteratively.
- *Approximate joint training:* This method considers the RPN and Fast R-CNN as one network during the training phase. During each SGD phase, the region proposals generated by the RPN are used to train the Fast R-CNN network and are treated as fixed, pre-computed proposals. The backward propagation works as usual and helps to compute the gradient of a combination of the RPN loss and the Fast R-CNN loss function described in section 3.2. The advantages of this solution are that it is easier to implement, since the network can be trained as a whole, without worrying about intermediate steps. The total training time is also reduced from 25% to 50% compared to the alternating training method. It is however ignoring the derivative with respect to the bounding box coordinates, which makes it approximate. A third method, *non-approximate join training*, that aims to correct this approximation is discussed but not implemented in the paper [28]. While this approximation should theoretically degrade the results somewhat, empirical results have shown comparable results between the two methods.

Faster R-CNN seems to be an efficient and practical solution for the object detection problem as it is the most advanced architecture among the different choices presented above [28].

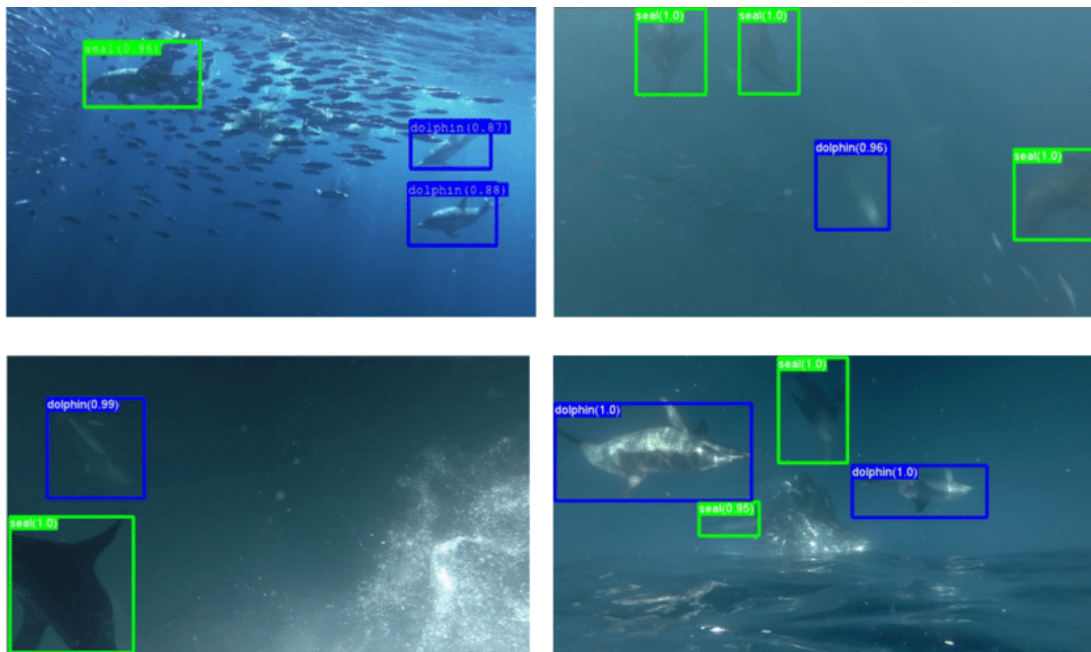


Figure 3.8: Examples of images processed by Faster R-CNN for the detection of marine predators [29].

### 3.4 Mask R-CNN

While Faster R-CNN and the earlier solutions tackle the problem of object detection, the Mask R-CNN architecture is built to offer a solution to the semantic segmentation problem. This refers to the process of classifying images into classes at a pixel level. Indeed, Mask R-CNN is capable of producing masks over the regions of an image occupied by the instances of the target classes. It is based on Faster R-CNN.

It still uses a RPN to propose candidate object bounding boxes. However, the second stage is modified to produce a binary mask for each RoI, in parallel with the prediction of the class and the bounding box offset. This is done by adding a mask branch based on the feature map alongside with the fully connected layers and the classification and regression sibling branches.

The RoI pooling layers is not sufficient to produce masks as this layer, by using quantization techniques, introduces misalignments between the RoIs and the

feature maps. To solve this problem, the mask branch actually contains a new layer called RoIAlign. This latter one removes the harsh quantization of the RoI pooling layers by aligning the extracted features with the inputs. The output of this layer is processed by a fully convolutional network (FCN), which is only composed of multiple (the number of layers depending on the backbone used) convolutional layers as shown in figure 3.9.

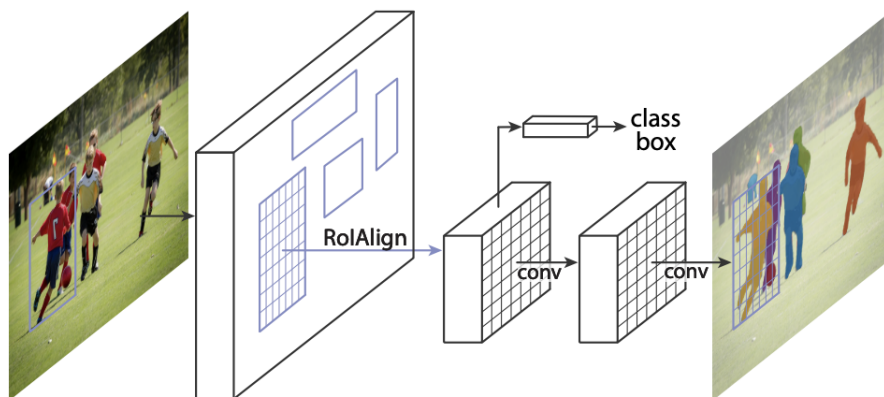


Figure 3.9: Mask RCNN architecture [30].

During the training phase the algorithm tends to minimize the following loss function:

$$L = L_{cls} + L_{box} + L_{mask}$$

where  $L_{cls}$  represents the classification loss and  $L_{box}$  the bounding-box loss as described in the previous section.  $L_{mask}$  corresponds to an average binary cross-entropy loss related to a per-pixel sigmoid function. This sigmoid function is applied on  $K$  binary masks of resolution  $m \times m$ . These  $K$  masks are applied to one RoI. Note that there is only one mask  $k$  used, related to the  $k^{th}$  class of algorithm. The floating number mask output is resized to the RoI size, and binarized at a threshold of 0.5.

With Mask R-CNN algorithm, unlike other state-of-the-art algorithms, the mask computation phase and the classification phases are independent. Thus the mask is generated without competing among classes. This means that once the observed object has been classified, the mask can be computed without concern for the categories.

Globally one can conclude that the performance of this algorithm is similar to the Faster R-CNN one in object detection but it offers a solution to semantic segmentation in parallel. [30].



Figure 3.10: Examples of images processed by Mask R-CNN [30].

### 3.5 Conclusion

The different architectures described in this section lead up to two principal solutions for our projects. The Faster R-CNN seems to be a good choice for object detection. It requires an annotated input composed of images and a bounding box and class label for each of the instances on these images for the training, and produces a bounding box, a label and a probability of certainty associated to each detected instance during the inference. Mask R-CNN is an alternative for object detection that also solves the problem of semantic segmentation. However, it requires an annotated mask over each instance of the input images for training instead of simple bounding boxes, and produces a mask, a bounding box, an associated label and a probability during the inference. As the advantages of Faster R-CNN match more accurately our needs in the scope of this project, we chose to use this architecture. However, we also use a Mask R-CNN prototype to automate the annotation task of our dataset. These choices are described in detail in chapter 5, along with the discussion of the advantages and disadvantages of both approaches with respect to our constraints.

# Chapter 4

## Insects applications

Several researchers have already deployed insect identification systems. These usually have two different goals. The first goal is to trap insects. There are several insect trap techniques that work differently depending on the species of interest. The second goal is to identify the trapped species. This part of the system can use the network architectures described in chapter 3. The section related to this goal enables us to make informed decisions when designing the trap prototype in our project, and gives us an idea of what can be achieved in insect detections with machine learning.

### 4.1 Insects traps

There are several popular insect traps used in entomology. This section aims at describing some of them.

The first trap is called the pitfall trap. This trap aims at capturing crawling insects. This consists of a container submerged in the soil that is filled with a preservative. When insects fall in this trap, they are blocked so that they cannot get out of the container.

The second one is called the Malaise trap. This is a tent-like trap made up with a mesh that captures flying insects. When the insects hit the mesh, they either fly upward or downward. Those that fly upward are funneled to the peak of the trap and reach a jar that contains some killing agent.

Another similar trap is the flight interception one. It is very similar to the previous one in the fact that it drives the insects into a jar filled up with preservative. The difference is that the mesh or the screen hit by insects is inclined so that they reach the jars at the bottom of the trap and thus are not driven upward.

The Lindgren Funnel Trap is able to capture beetle-like insects. It is made up of funnels placed on top of each others so that beetles can only move downward in

such a trap. At the bottom of the trap, below the funnels, a jar (that can contain ethanol for example) traps the insects.

One can also attract the insects with baiting. There are several different kinds of baiting. The baiting used is most likely to attract only a few insect species. For example, brown sugar and yeast can attract beetle, turpentine can attract some wood beetles, beer with molasses can attract longhorn beetles. Finally, wine and fermenting fruits can also attract ants, beetles, flies and so on.

Blacklight traps consist in projecting ultraviolet light on a blank sheet at night. Some insects that are active at night are attracted to these wave lengths lights.

Pan traps consist in container filled with soapy water. The soap is used to break the surface tension of the water so that the insects drown in water when they fall into the container.

## 4.2 Insects detection

Several image recognition techniques have already been implemented to count and identify pests in cultures. This section presents three different application examples. These applications are based on different objects detection algorithm presented in chapter 3, namely a support vector machine classifier, a CNN model and Faster R-CNN algorithm.

In the first project, an algorithm was developed to identify four different rice light-trapped pests in China. The system can capture the top and bottom images of each insects by two cameras to get more features. These features are based on color, shape and texture of insects. The algorithm was trained on four different species of Lepidoptera rice pests and achieved 97.5% average accurate rate. Once all the processing techniques are done, the features are input into a support vector machine classifier with radial basis kernel. This classifier is trained using 7-fold cross-validation. The researchers used an image database including as many non-target insects species as possible to reject the pictures of non-target insects [7].

Another project that uses cascaded convolutional neural networks have shown great results in a pest identification project in Taiwan (2018). The previous classical way to inspect and count insects involved traps with yellow sticky papers. The researchers used images of insects trapped on these taken with a Raspberry Pi camera. These images were used to train a model able to identify four different pest species. At first, the images are pre-processed so that they are segmented and the coordinates of the objects detected on the images are extracted. Then, the non-insects objects are distinguished from insects objects according to entomologists analysis. After this, 128x128 sized images in which there are some insects are cropped from the original one. These images are then split into a training and a testing set to build a CNN model. The architecture pipeline of this model is

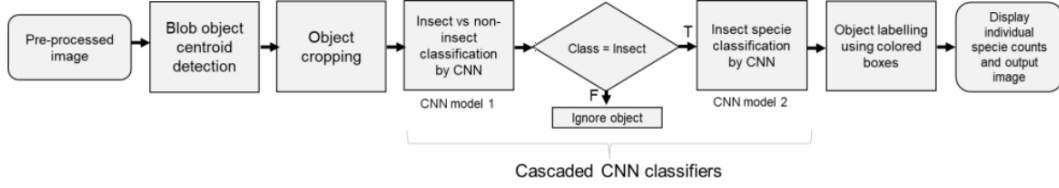


Figure 4.1: Architecture pipeline of the pest identification using CNN [6].

represented on figure 4.1. According to the results of the inference phase, the model is able to detect at least 93% of the insects present on the sticky papers and the average detection accuracy for the four species is around 86-92% [6].

Finally, the last application is an identification system that focuses on three different insects that damage the crops. The researchers that deployed this application made the comparison of three deep learning meta-architectures including Faster R-CNN architecture and focuses on three different insects species. This latter one performs with an accuracy of 95.33% and a sensitivity of 91.06%. The accuracy is the number of correctly classified insects divided by the total number of insects while the sensitivity or true positive rate is the measure of non-insect that are correctly classified. These metrics have already been defined in section 2.2 [5].

## 4.3 Conclusion

There exists several applications that aim at capturing insects and at identifying them. The insects traps usually focus on one or a few different insect species. Several researcher's teams have designed models to identify different insect species. These models usually focus on a small number of species (4 for instance) and perform well. The Faster R-CNN algorithm seems to outperform other more classical algorithms. This algorithm can lead to an accuracy of about 95% and a sensitivity of about 91% when it classifies four different classes.

These results confirmed our choice to use the Faster R-CNN architecture in the job presented in this thesis. Our work contains several added values with respect to the works presented above. Firstly, the main works that apply machine learning algorithm to insect species identification are deployed in Asia. This means that they do not focus on the European species that we can find in Belgium. Secondly, our final system focuses on ten different insect species. As seen, the presented works often track a small amount of insect species. Therefore, the total amount of species of interest considered in our solution seems to be an improvement with respect to the presented literature works.

## Part III

# Automated tracking of insects

### Contents

This part is the core of our master's thesis in which we mention the different goals and constraints, describe the methodology used, cite our different experiments and present our results in the two phases of the project: the model training, described in chapter 5 and the real time deployment of that model on a physical trap, which is described in chapter 6.

# Chapter 5

## Model training

This chapter describes the different steps that we took to produce the model intended for running on the trap. We first iterate over the different constraints that are associated to the training of the model, as well as the goals we are trying to achieve. We then describe the methodology that was used to achieve those goals. More specifically, we cover the architectural choices that we made, describe some implementation details and mention the expected limitations of the model. We then report the various experiments that we did, starting with the configuration of the first training and continuing with the research of the optimal hyper-parameters in the validation phase. Finally, we cite and interpret the results found in the previous sections to have an idea of the performance of our model.

### 5.1 Goals and constraints

The goal of this master’s thesis is to produce a system for the automated tracking of insects. More specifically, we aim to build a non-lethal insect trap able to take pictures and log autonomously the presence of the insects that it detects and its species. Indeed, the detected insects are associated with their perceived species, as long as it is among the different species of *Hymenoptera* that the algorithm recognizes (see table 5.1 for the list of these species). Appendix A offers a basic overview of the entomology concepts that are important for insect identification and detailed descriptions of the species handled by our algorithm. As time was a constraint for our work, we had to make some concessions on several points. For instance, the list of species had to be limited, but we also had to select relatively common species to maximize the number of occurrences in front of the trap and have as much relevant data as possible. The species were chosen according to the following criteria:

- The species must be part of the *Hymenoptera* order

- The species must be endemic to Belgium as this is where the trap is situated
- The species must be common
- We must find enough data to represent the species<sup>1</sup>

| Family          | Genus             | Species                    | Common name             |
|-----------------|-------------------|----------------------------|-------------------------|
| <i>Apidae</i>   | <i>Apis</i>       | <i>Apis mellifera</i>      | European honey bee      |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus hypnorum</i>     | Tree bumblebee          |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus hortorum</i>     | Garden bumblebee        |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus lapidarius</i>   | Red-tailed bumblebee    |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus lucorum</i>      | White-tailed bumblebee  |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus pascuorum</i>    | Common carder bee       |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus pratorum</i>     | Early bumblebee         |
| <i>Apidae</i>   | <i>Bombus</i>     | <i>Bombus terrestris</i>   | Buff-tailed bumblebee   |
| <i>Apidae</i>   | <i>Anthophora</i> | <i>Anthophora plumipes</i> | Hairy-footed flower bee |
| <i>Vespidae</i> | <i>Vespula</i>    | <i>Vespula vulgaris</i>    | Common wasp             |

Table 5.1: List of species that are recognized by the algorithm with their taxonomy.

As we aim to provide a generic solution for insect recognition, we do not have specific requirements for the system’s behavior during the recognition. Obviously, we want to maximize the precision and the recall of our classifier, but we do not especially have a preference for one over the other. We also need to keep in mind that this algorithm is intended to run on a physical device with limited computational power. This implies that we have to make concessions in the quality of the model to be able to run the inference of the captured insects in a reasonable amount of time. Another constraint relies on the fact that we do not have unlimited computational power (and time) for the training, validation and testing phases of the algorithm as well.

## 5.2 Methodology

To train the model, we used a specific methodology that is described in this section. At first, we describe our architectural choices. Then, we cover the implementation part of the model, which deals with its training and testing phases but also with the dataset creation and by the data augmentation and data preprocessing techniques.

<sup>1</sup>This is due to the fact that we did not have a prebuilt dataset at our disposal and that we had to construct it ourselves. This is discussed in the next section.

### 5.2.1 Architectural choices

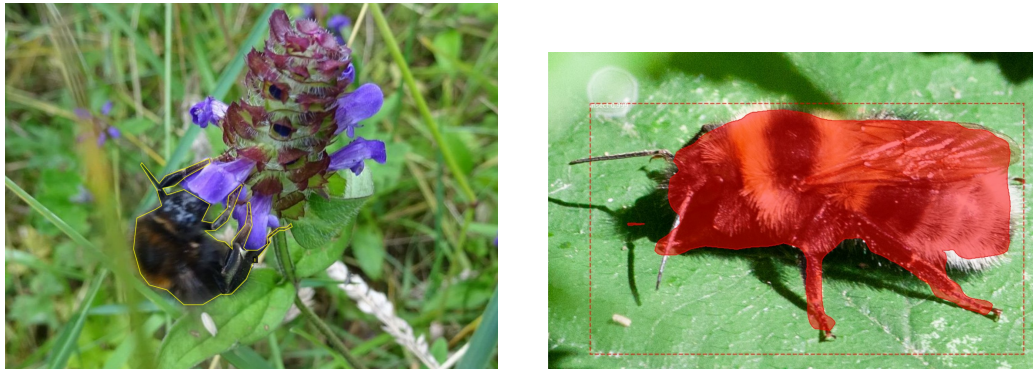
The expected behavior of the model defined in section 5.1 corresponds to solving the object detection problem for the set of classes of table 5.1. To do so, we tested two different strategies, which are the Mask R-CNN algorithm and the Faster R-CNN algorithm, that are described in chapter 3.

In theory, in contrast to Faster R-CNN, Mask R-CNN is able to find the precise edges of the detected insects, in addition to producing boxes around them. At first, we trained this model with a small dataset of 100 images for only four different classes. The results with such a dataset were not satisfying as the model could detect the presence of an insect in the image but could not precisely identify to which species it belongs to and could not precisely draw the mask around the insects. The solution to increase the performance of this model consists in building a larger dataset. However, to do so, it is necessary to manually draw the masks on each insect before the training. Due to time constraint, it is infeasible to do so for a large amount of images for each of the ten classes. Moreover, the final algorithm does not aim at finding the precise edges of each insect. It only aims at detecting an insect and finding its location.

Therefore, we decided to use Faster R-CNN algorithm for our final solution. This algorithm outputs bounding boxes around the recognized insects. With such boxes, it is possible to know the location of the insect accurately enough for our application. Furthermore, we used this algorithm because it seems to outperform the other current algorithms that build R-CNNs. As mentioned in section 2.1.3, there are several possible model's backbones such as VGG16 and ResNet50 (A residual network with 50 layers) for example. In practice, the residual network slightly decreases the losses of both region proposals networks and classifier networks compared to the VGG network. However, a model with ResNet50 architecture takes twice more time to train than a network with VGG16 architecture. As described in the section 5.3.2, we had to train a large amount of models to validate several parameters. Thus, we made the decision to use the VGG16 architecture. Moreover, we used the Adam optimizer [15] to update the stochastic gradient descent phase, as explained in section 2.1.2.

### 5.2.2 Implementation

This section describes the composition of the dataset and of the annotations, the training and validation procedures, the testing procedure and the implementation of additional preprocessing and augmentation methods in our project. The detailed functioning of the scripts, their interaction with the different files and folders, the structure of the project and the use of our project are described in the user manual in appendix B.



(a) Mask annotation

(b) Mask and bounding box prediction

Figure 5.1: Examples of annotation and prediction for the Mask R-CNN prototype.

## Dataset

The composition of the dataset was the first part of this project. As Faster R-CNN requires annotated images along with the bounding box coordinates of the instances of our classes, and that there is no existing dataset of that sort, we had to build it ourselves with respect to the constraints defined in section 5.1. The list of species presented in that section was constructed by analyzing the number of images available and by observing the common species of *Hymenoptera* in Belgium. The raw images (around 1500 images per class) were gathered from *observations.be* [31] and *gbif.org* [32]. All the images of insects in the figures of this section also originate from these two sources.

Though we have decided to use Faster R-CNN for the final application, we first experimented a bit with Mask R-CNN. Our first prototype was trained to recognize insects but was only composed of one class *insect* (beside the *background* class) and was able to produce bounding box coordinates around these insects quite accurately. That model was trained using a total of only 800 images of bees and bumblebees that were annotated manually. Mask R-CNN takes a set of points defining the shape of the insect, which was created using VGG Image Annotator (VIA) [33] and (as said in section 5.2.1) by drawing manually those shapes over each of the 800 images. As mentioned previously, the results produced by this model showed approximate masks but accurate bounding box coordinates as you can see on figure 5.1. The figure additionally shows an example of annotated images. This allowed us to produce the necessary annotations for the 15000 images used in the training, validation and testing of our final model. These automatically produced annotations were quickly reviewed, and we found that they didn't need any correction in most cases.

The final dataset is partitioned in 3 different sets: the training set containing

60% of the images that is used to train the model, the validation set with 20% of the images that is used to validate the model and finally the testing set containing the last 20% of the 15000 images used to evaluate the model.

The complete dataset used for the training of our models [34] and the Mask R-CNN implementation that we based ourselves on [35] are available on GitHub.

## Training procedure

Our implementation of Faster R-CNN, of the training and testing procedures are based on a pre-existing GitHub repository [36]. However, we made several structural changes compared to the jupyter notebooks of the original repository. This simple implementation did not have any support of validation, and we decided to add means to help the visualization of data and of the losses during the training. The GPU used for the training is the NVIDIA RTX 2080 card with 8Gb of memory.

The validation procedure is shown on figure 5.2. It starts by initializing the training configuration. Firstly, the images of the dataset are divided in three sets: the training, validation and testing sets. As mentioned above, the training set contains 60% of the images of the dataset, while the validation and testings sets both encompass 20% of them. It then sets the settings of the training (e.g. preprocessing methods and parameters).

The next step consists of producing a list of combinations of values for the different hyper-parameters that need to be validated. We build the exhaustive list of combinations from the list of parameters and the domains of their values.

The idea of the validation is to go over each of these combinations, to train a model with the training set, and to evaluate it with the validation set to be able to quantify the effect of the hyper-parameters. For each combination, the model's weights, the configuration and the optimizer are initialized according to the values of those hyper-parameters. Then it proceeds to a standard Faster R-CNN training, with a validation phase at the end of each epoch. More specifically, at the beginning of each epoch, it starts to iterate over all images in the training set, and for each of these images, it preprocesses or augments it according to the configuration, generates anchors and updates the weights of the RPN using that image and its annotation. The predicted result of the RPN for that image is processed and used to train the classifier. A training loss is computed for that image and used to update the running average loss for the entire epoch. Once all the images of the training set have been used, the same process happens with the images of the validation set, except for the fact that the weights of the networks are not updated anymore. This allows the computation of the average loss over the validation set, which helps avoiding overfitting and provides a basis on which the decision of the best combination of values for the hyper-parameters can be done.

Once the last epoch is over, the best validation loss (the minimal validation loss

over all the epochs) is kept in memory and the training for the next combination can start. At the end of the validation, the set of values for the hyper-parameters that corresponds to the lowest validation loss is considered optimal.

As it can be seen, we chose to use the approximate joint training method described in section 3.3. To understand more deeply how the configuration files are accessed, the losses are logged, or the images partitioned in the actual implementation, refer to appendix B.

## **Data augmentation and data preprocessing**

The different data augmentation and preprocessing techniques we chose to implement and validate is described in subsection 5.3.2. These methods essentially rely on image processing functions, and are implemented in our project as an additional step in the processing of the image before feeding it to the network. The correct operations are applied on the image according to the configuration.

## **Testing procedure**

Once the model is trained, we can evaluate it with different metrics that have been described in section 2.2.

If the model has been validated, the testing procedure must be done by only considering the images that were not used during the training phase neither the validation phase. This aims at avoiding overfitting. During the evaluation phase, we must apply the same preprocessing techniques to the evaluated images like the ones applied to train the model. These preprocessing techniques are described in subsection 5.3.2.

We also visually evaluated our results. Practically, the visualization routine consists in drawing the bounding boxes predicted by the model on the processed images. At first, it fits the image into the RPN. It then converts the output of the RPN into a RoI and fits it into the classifier model. This model outputs the bounding boxes corresponding to the detected classes on the image and the probability of their most likely corresponding label. Once the label and the boxes coordinates are known, it draws the boxes on the initial image and writes the labels on one edge of these boxes. To do so, we used methods implemented in the OpenCV library, described in subsection 2.3.3.

We also compute different metrics to evaluate the model. As in the visualization method, at first, the input image is processed into the RPN and into the classifier model. It evaluates the model by considering the AP score, the ROC AUC metric and the F1-score metric which are all defined in section 2.2. We use those metrics because we are trying to describe the model in a generic case since, as pointed out in our goals and constraints (see section 5.1), our final application does not

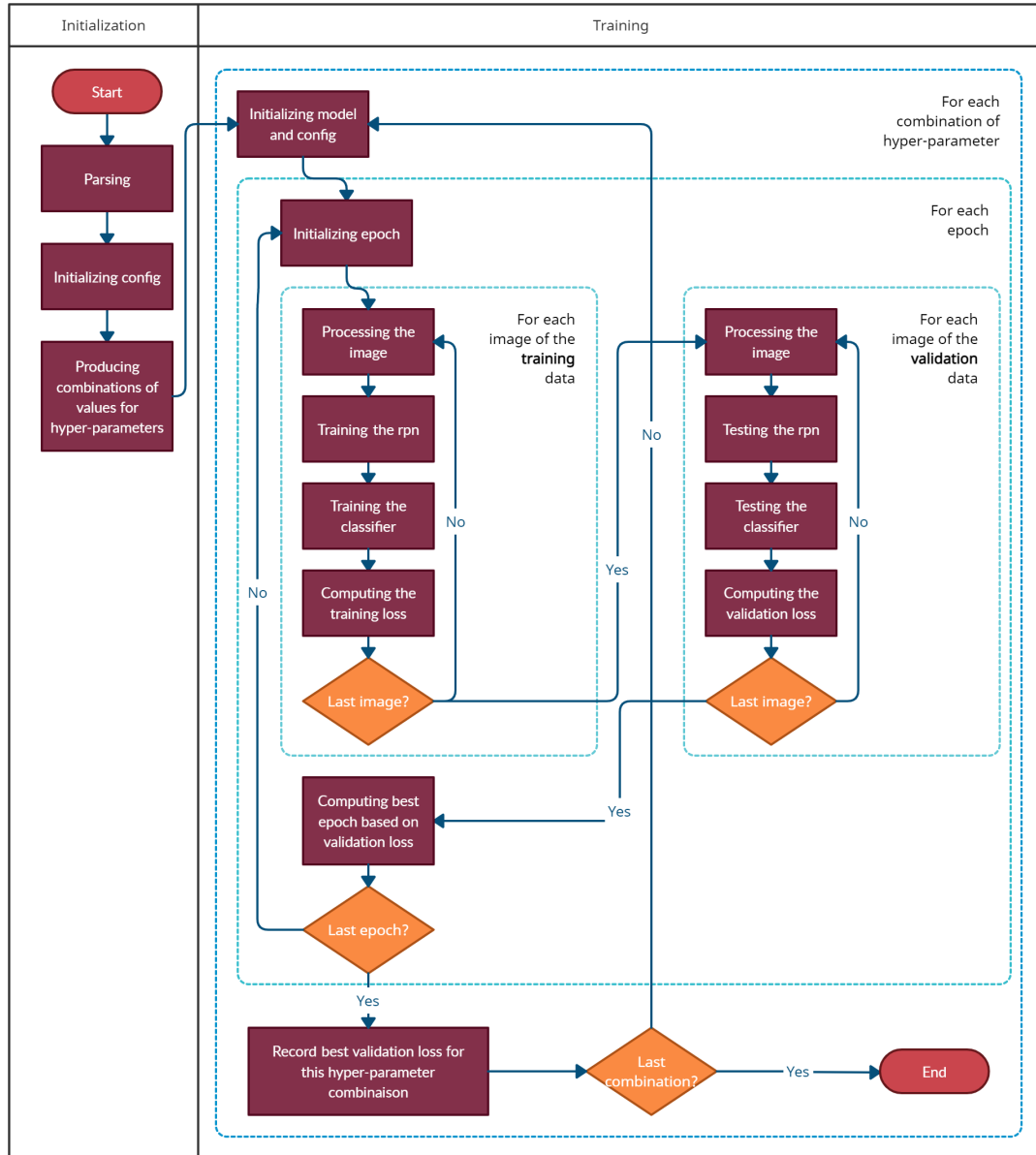


Figure 5.2: Training flowchart on which we can observe that we follow the approximate joint training method.

require favoring a metric over an other and the AP and ROC AUC metrics offer a good general view of the compromises that can be done between the different metrics with our model. We can also visualize those by plotting the precision-recall curve and the TPR-FPR curve. As the output of the model is not binary but can correspond to ten different classes, there are two different possible ways to compute these metrics.

The first way consists in evaluating the metrics for each possible class. By doing so, we consider the multi-class model as several different binary models which predict that an object is positive (for this class) with a certain probability. To do so, we only need to recover the most likely class corresponding to a bounding box and its corresponding probability. *For each possible class*, we then consider all the boxes labeled with this class and compare their probability with the actual label of the annotated bounding box, which is either 1 if the box surrounds the considered class or 0 if it surrounds another class. By comparing these values, we can then compute the different metrics mentioned before.

The second way consists in evaluating the model by considering all the different possible classes. To do so, we compute the micro average metrics, by considering each class with the same weight. Concretely, this is done by concatenating all the observations done to evaluate the metrics in the previous way. We then compute the metrics on these concatenate results.

Once the metrics are evaluated, it is also interesting to evaluate the best decision threshold. To do so, we made the assumption that both recall and threshold are important in our model's application. We did not want to give more importance to one of these metrics. So, we decided to choose the optimal threshold based on the F1-score metric, computed via the precision and recall metrics which are computed with all different classes. So, our application considers that a prediction is reliable once the model predicted a class with a probability that is larger than the decision threshold that leads to the maximal F1-score value.

### 5.2.3 Model limitations

The goal of the training part is to create a model that can be deployed on a large range of applications. Because of this, the model can be more limited in some specific applications than in others. We also chose to add some limitations constraints by ourselves so that it can fit several applications' architectures.

The developed model should be able to predict the different species on computer architectures that do not have big computation resources. So, the first limitation of the model is the inference time, which has to be as short as possible. As the inference time increases with the image size, we limited the input image size to 300x300 pixels. Thus, each input image is resized before it is used to train the model.

Moreover the training dataset also represents a limitation of the model. As the number of data images needed to train the model is very important and due to the rarity of insects during winter, it was impossible to create our own dataset by ourselves. Thus the dataset is composed of pictures taken by amateur or professional photographers. This means that one can expect the model’s performance to be better with the same kind of pictures. In other words, the model can be less accurate when processing images taken automatically (and with a fixed focus).

## 5.3 Experiments

To build the final model, we proceeded iteratively. At first, we made several choices of conception with respect to the final application’s goal. Then, we considered several successive simple models to validate preprocessing parameters’ values. This whole procedure is explained in this section.

### 5.3.1 Initial training

Before focusing on the ten different classes presented in table 5.1, we simplified the problem by training the algorithm to only distinguish four different classes, which are *Apis mellifera*, *Bombus hypnorum*, *Bombus hortorum* and *Anthophora plumipes*. This model was a first prototype of the final model. We chose to do so to evaluate the performance of a model with a certain amount of images in the dataset and some parameters that are important to consider when training the Faster R-CNN algorithm. The advantage of considering a simplified model is that it is easier to iteratively enhance it as its training is not as time-consuming as a model that considers all the classes. We fixed the values of some parameters during the initial training phase. We determined some other ones by observing the results of a validation phase. This is explained in subsection 5.3.2. This section only focuses on the choice of conception for the model training. During this phase, we did not apply any technique evaluated during the validation phase.

We chose to augment the data with three different techniques. When processing an image to input it in the network, for each of these techniques, there is a uniform probability that this augmentation technique is applied to the image. This means that a given image from the dataset can be modified from one epoch to the other. The two first augmentation techniques are horizontal and vertical flip. For each of them, there is one in two chance that an orthogonal symmetry is applied to the image. The third augmentation technique is a rotation of the image. The image is rotated by an angle randomly chosen in the following domain :  $\{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$ .

The second parameter we fixed is the range of possible values for the scales of anchor boxes, which are described in section 3.3. These anchor boxes correspond

to squares with a width of one of these possible values :  $\{64px, 128px, 256px\}$  and can be rescaled by the following possible factors :  $\{(1, 1), (\frac{1}{\sqrt{2}}, \frac{2}{\sqrt{2}}), (\frac{2}{\sqrt{2}}, \frac{1}{\sqrt{2}})\}$ . In those expressions of the shape  $(x, y)$ ,  $x$  represents the horizontal scaling factor and  $y$  represents the vertical scaling factor. This means that for a scaling expression  $(x, y)$ , a square of width  $w$  is rescaled as a rectangle of width  $x * w$  and of length  $y * w$ . The important condition to be respected is that the initial square width should be lower than the width of the input rescaled image, which is 300 pixels.

The three channels of the image are recentered around 0 by subtracting a constant chosen empirically.

There are also some parameters linked to the convolution and pooling phases. These phases are configured to process four different regions of interest at once and the stride value is set at 16 pixels.

As mentioned in section 3.3, the algorithm considers its predicted boxes as positive and negative by observing their IoU values. We parameterized the algorithm such that a predicted box with an IoU greater than 70% is considered as positive and such that a predicted box with an IoU lower than 30% is considered as negative. The boxes with an IoU value between these two margins are considered as neutral.

As mentioned in section 3.2, the classifier network can learn which region proposals correspond to the background and which ones correspond to a class known by the algorithm. To do so, it computes the IoU between each ground-truth box and its best corresponding predicted box. It then attributes a label to the predicted box only if its corresponding IoU value is greater than 50% and the predicted box is skipped if its corresponding IoU value is lower than 10%.

The number of epochs for the training was determined dynamically by observing the validation loss after each epoch. The weights of the model were only saved when the validation loss decreased (hit a lower value than the lowest encountered earlier). That way, we could observe the optimal number of epochs that prevents overfitting, and use it as a scale for the many training that were made during the validation and that is explained in the next section. The evolution of the validation loss is shown on figure 5.3, and the optimal number of 21 epochs was found. Indeed, after epoch 20 (the number of epochs starts at 0), the validation loss seems to stagnate around the same value (with fluctuations between each epoch) before starting to go up, showing the start of overfitting.

### 5.3.2 Validation

The validation had for goal to minimize the loss of different models built with all the possible values combinations of hyper-parameters. This loss metric is computed over the validation set, using the 4-classes architecture described in the previous section. The built models are then compared to each other in order to select an optimal value (that minimizes the validation loss) for each hyper-parameter. Due

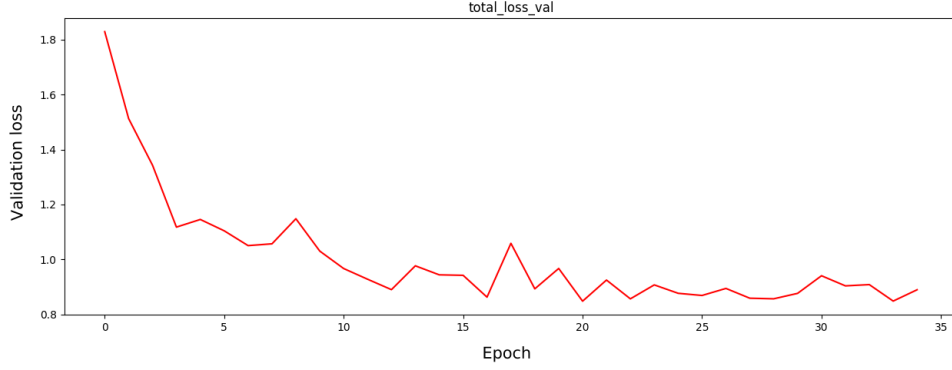


Figure 5.3: Evolution of the validation loss during the initial training of model6000 (epoch number starts at 0).

to the time constraint defined above in section 5.1, all these models are trained on a smaller dataset which is built and evaluated as a set of 200 images per class, randomly selected from our complete 4-classes dataset. This allowed the different training phases to be done faster as a validation of all the combinations of parameters on the complete dataset would have been unfeasible. However, to improve the statistical quality of our results, we ran this whole operation four times, and based our decisions on the average and median loss values.

In this work, the validation phase only focuses on preprocessing techniques and data augmentation meta parameters. The preprocessing techniques tested are briefly described below.

## Hyper-parameters

- **Noise reduction**

Two different noise reductions techniques were tested on images, namely the box-filtering technique and the Gaussian blur application technique.

The box-filtering technique consists in removing the noise in the data caused by the changes in luminosity. This method applies a convolution filter on the input image. It takes the average of all the pixels under kernel area and replaces the central element with this average. The kernel size was chosen according to experimental results that were produced with different parameters. Higher kernel sizes seemed to increase the best validation loss of the model, so the shape of  $(2, 2)$  was chosen [37].

The Gaussian blur filter technique also aims at reducing the noise of an image.

In contrast with the box-filtering technique, the coefficients of the convolution filter do not have equal coefficients. The image is convolved with a Gaussian kernel, which is a kernel that has coefficients which correspond to the outputs of a Gaussian function. The Gaussian kernel applied in the actual validation has a square shape of 7 pixels per side [38].

- **Histogram equalization**

An image can be represented as an histogram, which plots for each tonal value the number of pixels with this tonal value. The tonal value which corresponds to the brightness of a pixel. Histogram equalization is a technique that increases the global contrast of an image, especially when the image is represented by close contrast values. With this adjustment, the intensities can be better distributed on the histogram [39].

- **Gamma correction**

This technique is used to encode and decode luminance<sup>2</sup>. In practice this is done by converting each pixel value  $I$  from each of the three Red-Blue-Green channels that represent the image into a new value  $I'$  using equation 5.1 where  $\gamma$  is a value to be determined by validation. For  $\gamma = 1.0$ , the pixel values of the image are conserved. In practice, the value of  $\gamma$  was chosen with an independent validation phase during which the other parameters were not changed. This small validation procedure led to an optimal  $\gamma$ -value of 4.0. An example of gamma correction is shown on figure 5.4 [41].

$$I' = 255 * \left(\frac{1}{255}\right)^\gamma \quad (5.1)$$

- **Brightness jitter**

The brightness jitter technique is a data augmentation technique that aims to apply some luminosity changes on the input image to produce a new image. This allows the network to be able to learn the invariance of brightness in the case of insect detection and recognition. A small change in brightness is applied to the image before being processed by the network. The amplitude of that change follows a uniform distribution defined between the bounds  $(-\beta, \beta)$ . In practice, the value of  $\beta$  was determined during an independent validation phase during which the other parameters were not changed. This small validation procedure led to an optimal  $\beta$ -value of 0.2.

---

<sup>2</sup>Luminance: Photometric measure of the luminous intensity per unit area of light travelling in a given direction in an image [40].

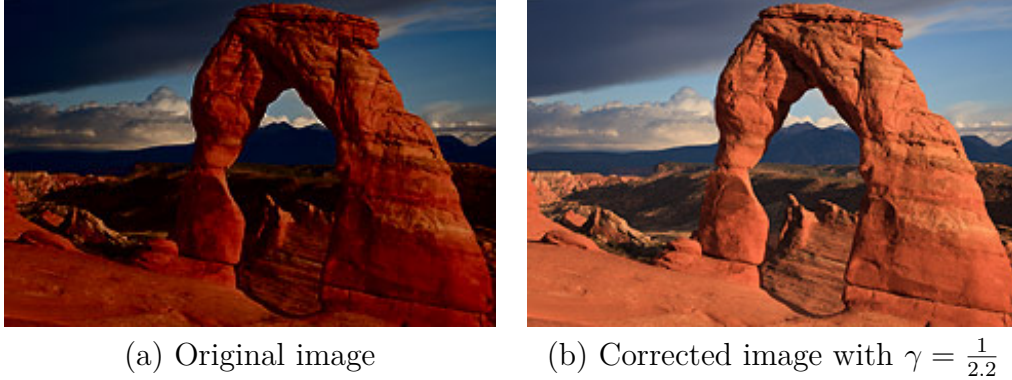


Figure 5.4: Images before and after gamma correction [42].

### Optimal set of hyper-parameters

To evaluate the best preprocessing phase to be applied to each image, we considered all the possible combinations of the hyper-parameters' values and validated *model800*, which corresponds to the model built and evaluated with a dataset of 200 images in each of the four different classes. Indeed, we made the assumption that the best parameters' sets for *model800* should lead to good results for *model6000* too. The table 5.2 presents all these combinations and their corresponding code. To validate this model, we selected the four different parameters' values set that tend to minimize the best validation loss obtained through the epochs. To do so, we repeated the validation procedure four times. The results of the four validation steps are presented on figure 5.5 that presents one error bar for each parameter set. The upper extremity of an error bar corresponds to the highest best loss value obtained during the four different phases. On the contrary, its lower extremity corresponds to the lowest best loss value obtained during these phases. The dot on each error bar corresponds to the average best loss value among the phases. As one can see, the parameters sets number 13, 14, 15 and 16 tend to minimize the loss metric. As referred in table 5.2, those parameters sets correspond to all the possible combinations obtained by applying or not the brightness jitter augmentation technique and the Gamma correction preprocessing technique. Note that the parameter set number 21 also leads to a small average loss value but we did not retain this set because of the high variance of the loss value across the validation phases.

| Code | Histogram | Noise Reduction | Brightness Jitter | Gamma Correction |
|------|-----------|-----------------|-------------------|------------------|
| 1    | True      | None            | True              | True             |
| 2    | True      | None            | True              | False            |
| 3    | True      | None            | False             | True             |
| 4    | True      | None            | False             | False            |
| 5    | True      | Gaussian        | True              | True             |
| 6    | True      | Gaussian        | True              | False            |
| 7    | True      | Gaussian        | False             | True             |
| 8    | True      | Gaussian        | False             | False            |
| 9    | True      | Box Filter      | True              | True             |
| 10   | True      | Box Filter      | True              | False            |
| 11   | True      | Box Filter      | False             | True             |
| 12   | True      | Box Filter      | False             | False            |
| 13   | False     | None            | True              | True             |
| 14   | False     | None            | True              | False            |
| 15   | False     | None            | False             | True             |
| 16   | False     | None            | False             | False            |
| 17   | False     | Gaussian        | True              | True             |
| 18   | False     | Gaussian        | True              | False            |
| 19   | False     | Gaussian        | False             | True             |
| 20   | False     | Gaussian        | False             | False            |
| 21   | False     | Box Filter      | True              | True             |
| 22   | False     | Box Filter      | True              | False            |
| 23   | False     | Box Filter      | False             | True             |
| 24   | False     | Box Filter      | False             | False            |

Table 5.2: All possible combinations of values for the validation phase of model800 and their corresponding code.

Once we validated the best parameters for *model800*, we started the validation of another model called *model6000*, also restricted to four different classes but built and evaluated with a dataset that contains 1500 images for each class, which corresponds to the number of images per class of the final model with 10 classes. Thus we made the assumption that the set of parameters that leads to the minimal loss value with the four classes model probably leads to the minimal loss value of the ten classes model. Moreover, we validated *model6000* with only two validation phases due to time constraints. The results of these validation phases are presented with the error bars plot on figure 5.6. As one can observe, the set of parameter number 14 tends to minimize both mean and variance of the best validation loss

value computed across the validation phases. This set only corresponds to the application of the brightness jitter augmentation technique.

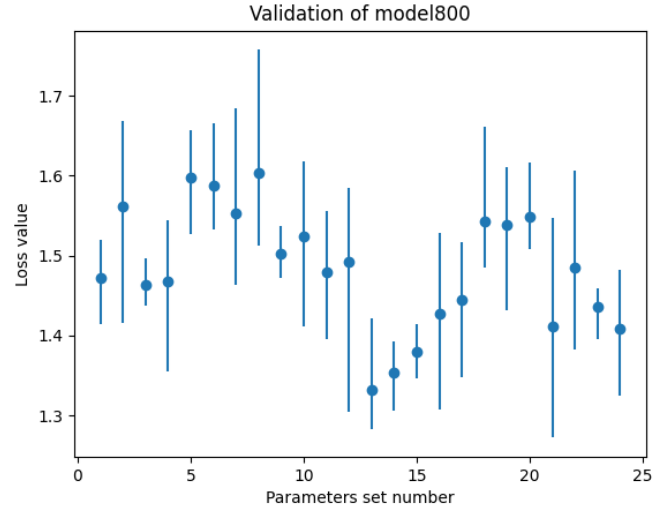


Figure 5.5: Validation phase of a model built and evaluated with 200 images in each of four classes.

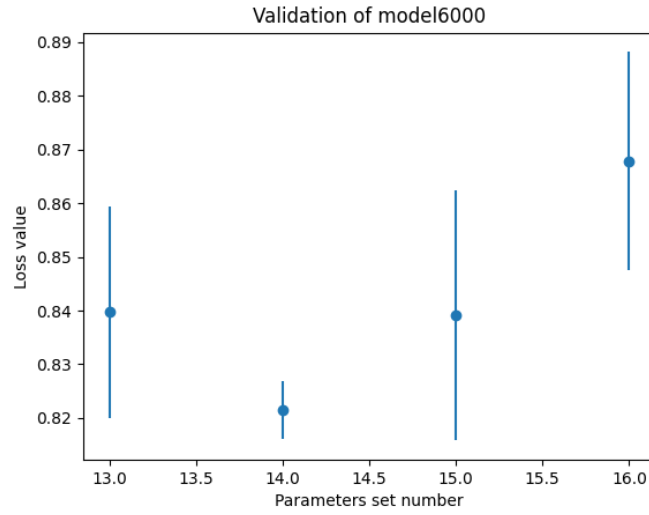


Figure 5.6: Validation phase of a model built and evaluated with 1500 images in each of four classes.

This leads us to the conclusion that the model built with 10 different classes

should perform better if we augment the data with the brightness jitter technique besides all the augmentation techniques applied to the images that are described in subsection 5.3.1.

## 5.4 Results

The validation process is based on the total loss of both RPN and classifier networks. However, this loss metric does not give information on the precision or the recall of the model. We thus evaluate the best model obtained during the validation phase by considering several different evaluation metrics that are relevant with respect to the goal of our application, in order to assess its quality. We also discuss the choice of the detection threshold and its impact on the different metrics of the model.

### 5.4.1 Evaluation of the model

At first we compute the metrics evaluated on each class separately, which means that we evaluate the model as a set of binary classifiers, corresponding to one class each.

The table 5.3 presents the summary of the computed ROC AUC and AP metrics for each of the ten classes. We can notice that the AP values related to the *Apis mellifera* and to the *Anthophora plumipes* which are respectively 0.878 and 0.868 are well higher than the AP of the other classes. This can probably be explained by the fact that these classes are the only *Apis* and *Anthophora* species and thus present distinguishable characteristics with respect to the other classes. One can also notice that the *Bombus lucorum* and the *Bombus terrestris* classes present lower AP values of 0.526 and 0.602 respectively compared to the other classes.

| Class                      | AP    | ROC AUC |
|----------------------------|-------|---------|
| <i>Apis mellifera</i>      | 0.878 | 0.818   |
| <i>Bombus hypnorum</i>     | 0.756 | 0.739   |
| <i>Bombus hortorum</i>     | 0.778 | 0.778   |
| <i>Bombus lapidarius</i>   | 0.624 | 0.608   |
| <i>Bombus lucorum</i>      | 0.526 | 0.547   |
| <i>Bombus pascuorum</i>    | 0.778 | 0.740   |
| <i>Bombus pratorum</i>     | 0.683 | 0.665   |
| <i>Bombus terrestris</i>   | 0.602 | 0.647   |
| <i>Anthophora plumipes</i> | 0.868 | 0.853   |
| <i>Vespula vulgaris</i>    | 0.786 | 0.796   |

Table 5.3: ROC AUC and AP metrics per class for the final model.

The figures shown in table 5.3 quite resemble the expected results as the classification can be quite complex in the case where the species of insects we consider look quite alike and the images are captured in a complex environment. There is no common plain background, instead the grounds and the vegetation behind all insects are quite diverse. The position of the insects on the images also play an important role during the detection as similar species might be distinguished only using a small amount of characteristics that might not always be visible in certain positions or with a certain luminosity. It is also expected to observe a slightly degraded quality in our model that is due to the fact that we used an input image size of 300x300 (instead of 500x500 in the original Faster R-CNN paper [28]) in order to reduce the training and inference time.

The weak performance of the worst two classes could be explained by different hypotheses. The first one is the fact that the histogram of pixels intensity is not distributed in the same way as for the other classes, as we can notice on figure 5.7. The graphs presented on this figure represent the distribution of the average pixel intensity of an image converted in gray scale. This graph was built by only using the training dataset. The histograms related to all classes except the *Bombus lucorum* and *Bombus terrestris* are distributed in a similar way as *Bombus hypnorum* (below graph on the figure). The two exceptional classes both present a large number of images with a high average gray intensity value. This means that training images corresponding to these two classes are on average lighter than the images related to the other classes.

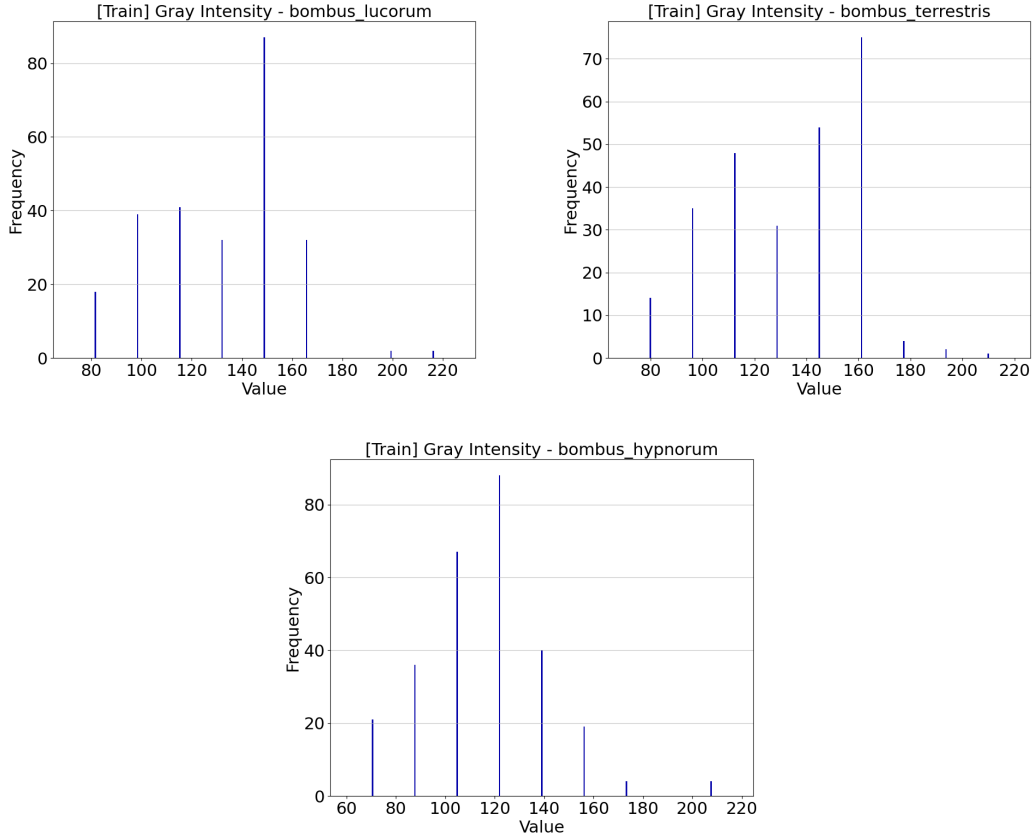


Figure 5.7: Gray intensity distribution for the training set of *Bombus terrestris*, *Bombus lucorum* and *Bombus hypnorum*.

It could also be explained by the fact that the images of *Bombus lucorum* and of *Bombus terrestris* correspond to close-up views of the insects, on the contrary with the other classes. Indeed as one can see on figure 5.8, which represents the repartition of the number of instances according to the ratio of the area of a bounding box with respect to the area of the whole image. Again, on this image, the diagram for the *Bombus hypnorum* class is similar to the diagrams corresponding to the other classes. As we can see on the diagrams related to *Bombus lucorum* and *Bombus terrestris*, the images of these classes tend to more often present a big ratio of the area of a box with respect to the area of the whole image.

Both these hypotheses imply that the dataset could be adapted to conform those classes to the others and in order to observe results that are in a closer range to the results of those other classes.

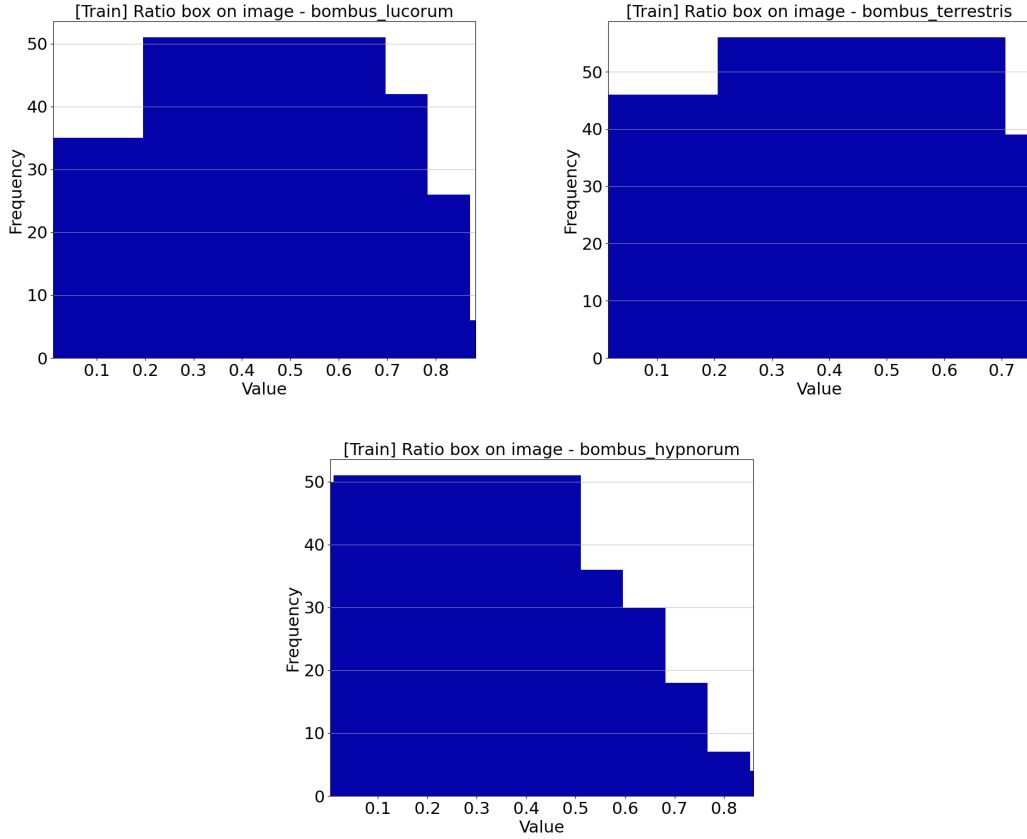


Figure 5.8: Distribution of the ratio of the bounding box area on the image area for the training set of *Bombus terrestris*, *Bombus lucorum* and *Bombus hypnorum*.

Finally, to analyze the performance of the model, it is also useful to observe the confusion matrix, presented on figure 5.9. The horizontal axis of the matrix corresponds to the actual labels and the vertical axis of the matrix corresponds to the labels predicted by the model. The mapping between the indices of the matrix and the classes names is shown in table 5.4. As we can see, the model tends to confuse the *Bombus lucorum* and the *Bombus terrestris*, which explains the lower performance related to these two classes. Actually, in practice, to distinguish these two species, entomologists tend to use a microscope or deoxyribonucleic acid (DNA) analysis [43]. Thus, it is not surprising that the model struggles to distinguish them.

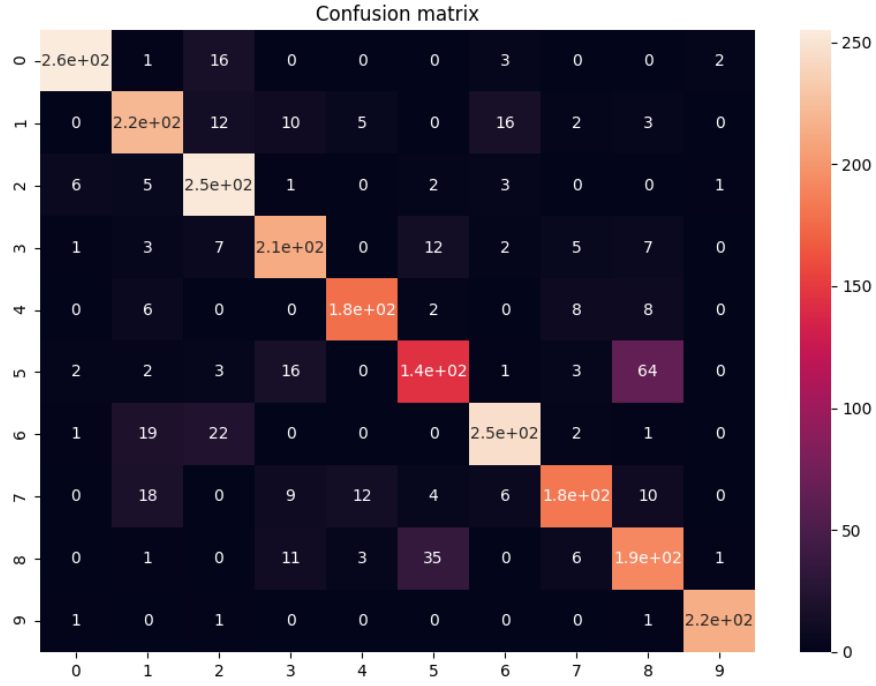


Figure 5.9: Confusion matrix of the model with 10 classes.

| ID | Class                      |
|----|----------------------------|
| 0  | <i>Apis mellifera</i>      |
| 1  | <i>Bombus hypnorum</i>     |
| 2  | <i>Anthophora plumipes</i> |
| 3  | <i>Bombus hortorum</i>     |
| 4  | <i>Bombus lapidarius</i>   |
| 5  | <i>Bombus lucorum</i>      |
| 6  | <i>Bombus pascuorum</i>    |
| 7  | <i>Bombus pratorum</i>     |
| 8  | <i>Bombus terrestris</i>   |
| 9  | <i>Vespula vulgaris</i>    |

Table 5.4: Mapping of the indices of the confusion matrix to the class represented.

Note that for each class, a histogram built with the training set is very similar to its corresponding histogram built with the testing set. This means that the training

images are similar to the testing images, for all classes. With this property, we are sure that the model does not underperform because of dissimilarities between the training and the testing set.

### 5.4.2 Evolution of the metrics for different thresholds

In this subsection, we observe the evolution of the metrics for different thresholds in order to understand the behavior of the model. The metrics presented in this subsection are computed on all the different classes at once (using a micro average). This means that the contributions of all classes are aggregated to compute them.

The precision-recall curve related to all classes is plotted on figure 5.10 (a). The area under this curve corresponds to the AP score of the model, all classes taken into account, which is equal to 0.739. As one could expect by observing the convexity of the curve, this value is high, which means that the model keeps a relatively high precision for increasingly high values of recall. We can observe that the best recall value we can achieve is around 80%, however, that comes at the cost of a degraded precision value of a bit more than 60%. On the other hand, we can choose to sacrifice the recall to achieve really high precision values. As each pair of precision-recall corresponds to a threshold value, it makes sense that the choice of that threshold is really important to find a good trade-off between the precision and the recall.

It is also important to discuss the reason why our model cannot achieve a better recall than 80%. It is due to the fact that we are not only assessing a classifier but the whole Faster R-CNN. Indeed, in the cases where the RPN is not able to predict a bounding box over an instance of a class, this instance is not classified, whatever the threshold is. Even with a threshold of 0, these *lost* instances are not classified and we do not obtain a recall of 100%. However, this is totally expected in a Faster R-CNN architecture as the RPN also had to learn and is not perfect. Thus, the optimal recall value of 80% is an evidence of the flaws of the RPN, and helps us to assess its quality.

The precision is nevertheless quite high for all recall values, compared to the precision of a random 10-classes classifier that would be hovering around 10%. This is why we can consider our model effective though it is not perfect. Possible improvement paths are discussed in section 7.1, but the model can be used for the detection and classification of insects and provides results of a relatively good quality.

We also computed the ROC AUC value based on all classes too, which is 0.719. This value corresponds to the area under the curve plotted on figure 5.10 (b). Here again, we notice that increasing the TPR comes at the cost of increasing the FPR. For example, we can be really close to a FPR of 0 with a TPR of 0.4, but we can reach a TPR value of around 0.7 for a higher value of FPR of 0.5. Again, the

choice of the threshold is responsible for finding a correct TPR-FPR pair.

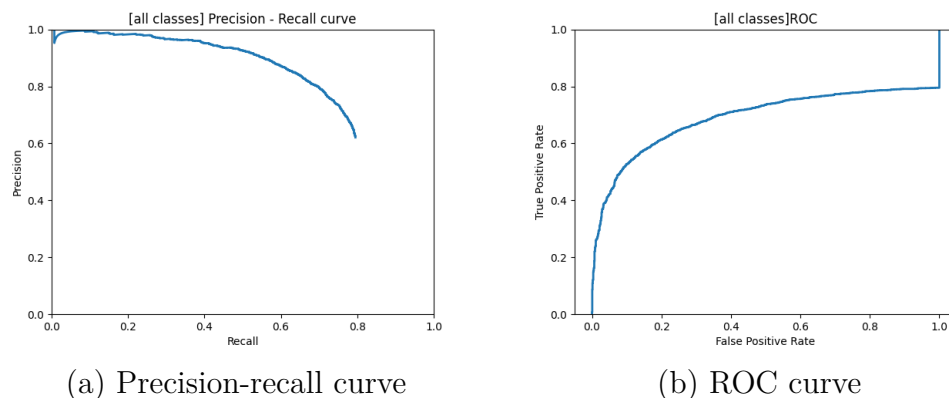


Figure 5.10: Evaluation curve with respect to all classes.

### Optimal threshold

As mentioned above, the choice of the threshold is crucial for obtaining a good trade-off between the different metrics. As our goals and constraints do not favor some metrics over others, we chose to maximize the F1-score in order to obtain a correct trade-off between the precision and the recall.

The probability threshold that maximizes the F1-score computed on all classes is 0.6. This threshold value leads to a F1-score of 0.754. This means that the classifier part of the final model ignores all the predictions where the probability that the box is related to a specific are strictly lower than 0.6. This minimum acceptability threshold maximizes both prediction and recall. This property is verified on the graph of figure 5.11. Indeed, with a threshold of 0.6, the model has a precision of 0.768 and a recall of 0.741.

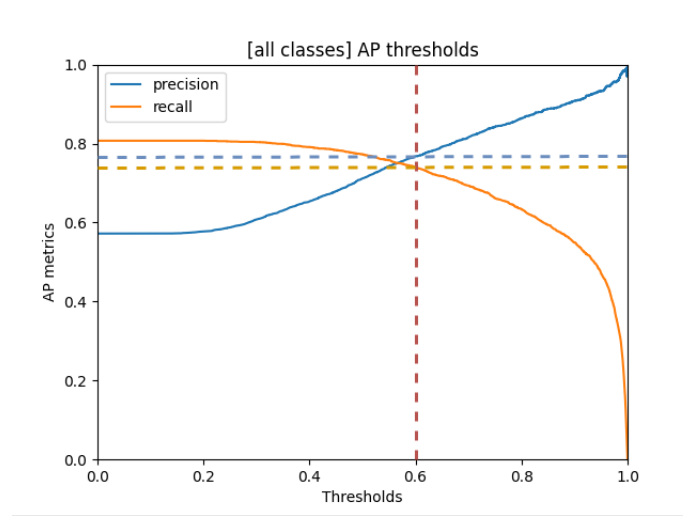
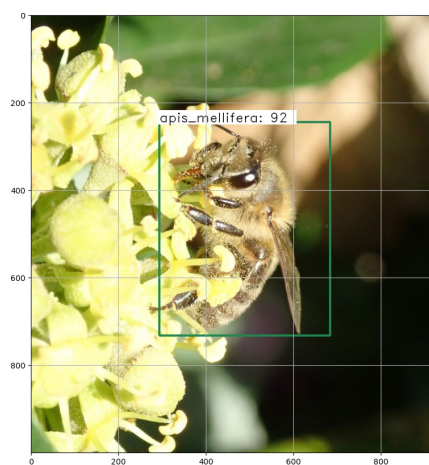


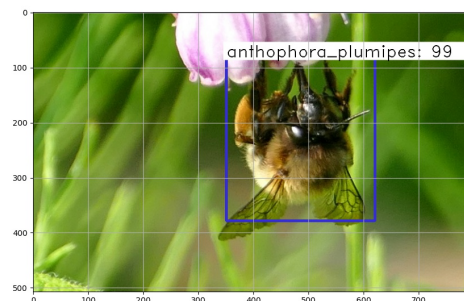
Figure 5.11: Precision and recall values related to all combined classes depending on the probability thresholds.

### 5.4.3 Examples

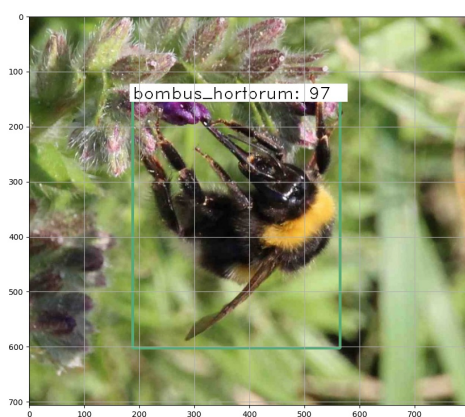
This section aims at presenting some good classifications of the algorithm. These classifications are presented on figures 5.12 and 5.13. Each processed picture contains a bounding box labeled with a species name and a number. This corresponds to the most probable species predicted by the model and the probability, expressed as a percentage, that the insect well corresponds to this species. As we can see on these examples, our model is often capable of identifying and correctly classifying with a probability that is often above 90%.



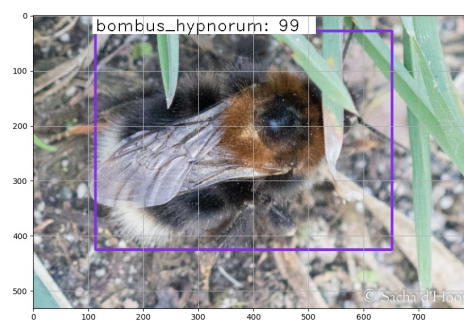
(a) *Apis mellifera*



(b) *Anthophora plumipes*

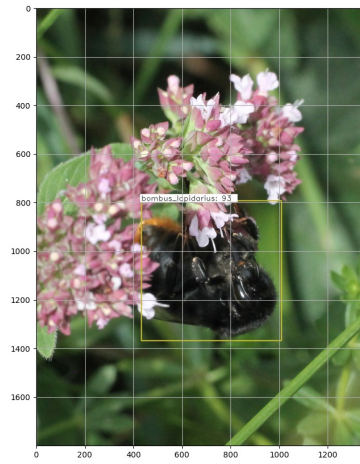


(c) *Bombus hortorum*

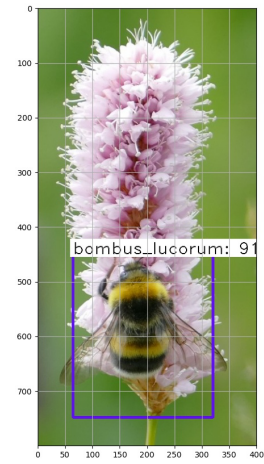


(d) *Bombus hypnorum*

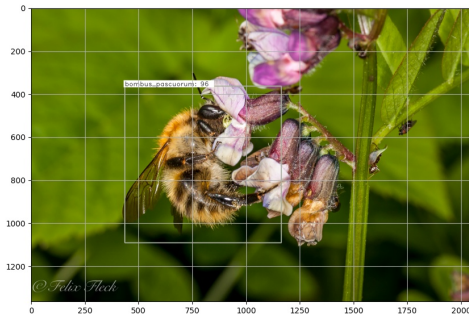
Figure 5.12: Examples of algorithm prediction - part 1.



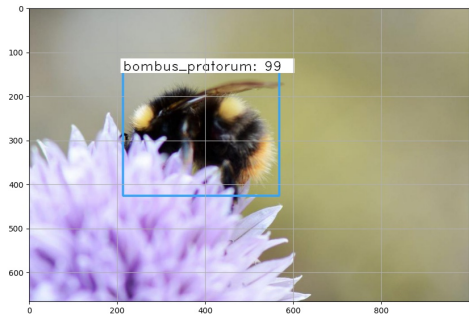
(e) *Bombus lapidarius*



(f) *Bombus lucorum*

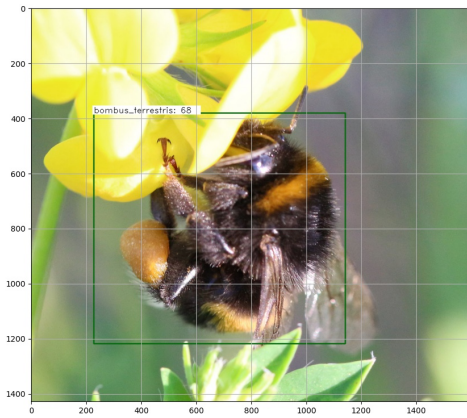


(g) *Bombus pascuorum*

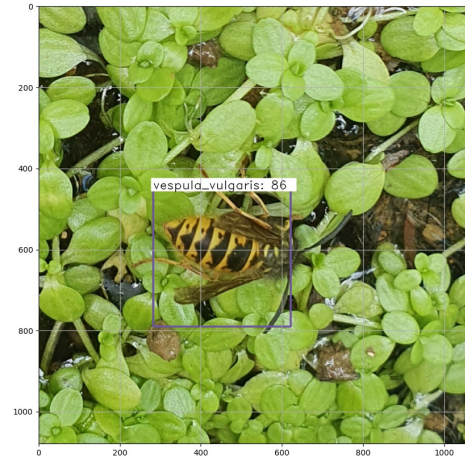


(h) *Bombus pratorum*

Figure 5.13: Examples of algorithm prediction - part 2.



(i) *Bombus terrestris*



(j) *Vespula vulgaris*

Figure 5.14: Examples of algorithm prediction - part 3.

## 5.5 Conclusion

The first important part of the project consists in training the best model able to recognize ten different *Hymenoptera* species. To do so, we made several choices of conception such as the use of Faster R-CNN with VGG16 for the backbone. Then we successively considered two different datasets to perform the validation phase, which focuses on four different preprocessing techniques. These two datasets contain four different classes. The first one includes 200 images per class. This dataset aimed at validating the model with all possible combinations of preprocessing parameters. The second dataset includes 1500 images per class, like the final 10 classes model. This dataset was used for the validation of the four best parameters set obtained with the validation through the first dataset.

After the validation phase, we evaluated the final 10 classes model with the optimal hyper-parameters by considering its AP score and its ROC AUC. These metrics considering all the different classes are respectively 0.739 and 0.719. We proceeded to find an optimal detection threshold that maximizes the F1-score of the model, and chose the value of 0.6.

While the recall is limited by the quality of the RPN, the model is still able to reach a high precision for the classification. The quality enables us to use this model to produce relatively accurate detections of insects and allows us to proceed to the next chapter for the use of the model on images produced in real-time.

# Chapter 6

## Real-time deployment

This chapter describes the insect trap on which the model validated and presented in the previous chapter is deployed. We first enumerate the precise goals of this application and its constraints. We then focus our explanations on the autonomous trap itself, namely the used method to attract insects and the electrical circuit. Next, we describe the code deployed on the Raspberry Pi. This program has for goal to detect movement, crop the insect region and launch the inference of our model on it. We finally conclude the chapter with a quantitative and a qualitative analysis of the results and the performance obtained during the experiment phase, and discuss some possibilities of improvement.

### 6.1 Goals and constraints

The final goal of this thesis is to deploy the algorithm presented in chapter 5 on an autonomous insect trap equipped with a camera. This trap should thus be able to attract the *Hymenoptera* species, detect them in real-time and identify them using the trained model. To achieve this goal, we had to respect several limiting constraints.

The first important constraint is that the hardware on which the algorithm is deployed to run the inference should be small enough to fit into the trap but also be powerful enough to reduce the inference time. We also have to try to minimize the budget of the trap. We worked with a Raspberry Pi 4, but it implies some constraints:

- The Raspberry PI 4 has a limited amount of random access memory (RAM) (4GB), however this should not be a limiting factor in our case.
- The storage space is limited, meaning that we need to manage space efficiently.

- The GPU memory is very limited compared to other hardware such as the Jetson Nano that are designed to support neural networks. Hence, the computation will exclusively rely on the CPU of the Raspberry Pi 4.
- The CPU power is limited, and will be the limiting factor of the inference.

Moreover, we wanted the whole system to be autonomous so that it can be deployed anywhere. To do so, the trap is powered from a battery that is charged with solar panels. This system has to be able to run during a whole day and can automatically shutdown at night.

Finally, the quality of the pictures taken by the camera are supposed to be as good as possible to increase the performance of the identification model. By that, we mean that the images taken must resemble the images of our dataset as much as possible, though they are taken automatically and with a fixed focus. This is quite important to ensure that the quality of the model assessed in the previous chapter is assured.

## 6.2 Autonomous trap

This subsection describes the way our model is used on the physical trap to make detections and classifications of insects. As stated in the constraints, there are limitations associated with the use of the Raspberry Pi 4. We assess the impact of these limitations in the deployment of our model on the embedded hardware of our trap and detail our methodology to alleviate the problems that these constraints caused.

### 6.2.1 Trap layout

In this subsection, we describe our design choices. In the first part, we explain how we proceed to reach the goal of the trap, which is attracting the insects to take pictures of them. In the second part, we justify the selection of our electrical components to make the trap autonomous.

#### Attracting insects

The goal of the application is to automatically take pictures of insects by using a Raspberry Pi. To do so, we had to imagine a trap that is able to attract the insects just in front of the camera. As the dataset we used to train the model is composed of pictures taken by professional and amateur photographs, these ones usually are close-up views of insects. Thus, it means that we have to attract the insects right in front of the camera, and thus the area captured by the camera

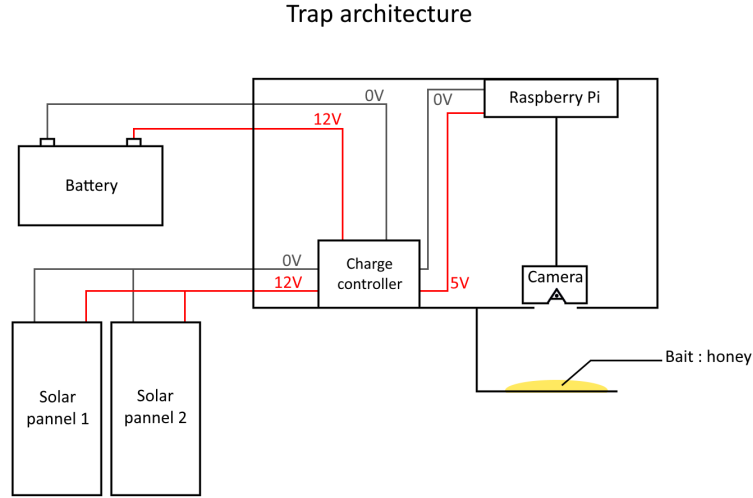


Figure 6.1: Scheme of the deployed insect trap.

should be very small. Therefore, as the insect's photographs are taken close to the lens, the focus has to be very accurate. Indeed, a bad focus could lead to a bad image quality and thus reduce the performance of the system.

To attract the insects, we built a trap by following the advice of Robert Lequeux who is a beekeeper. The trap is composed of a platform placed 15 centimeters in front of the camera. To attract the *Hymenoptera* species on this platform, we painted it in blue and put some honey on it. Indeed, the bees and bumblebees have the particularity to only distinguish blue and yellow colors and they are attracted by honey. The architecture of the trap is presented on figure 6.1. It is important to notice that the trap and the battery are completely sealed and thus resistant to the weather conditions.

## Electrical circuit

We wanted the system to be completely autonomous. So, in addition to autonomously detect and identify the *Hymenoptera* species, it also has to be powered up by renewable energy. As the insect species we focus on only live at day, the most simple way to keep the system alive consists in using the solar energy. As represented on figure 6.1, the solar panels recharge a battery to power the Raspberry Pi. This power supply is managed by a solar charge controller. This controller has two main different roles in such a system. Indeed, if the solar panels provide more electricity power than needed by the Raspberry Pi, the solar panels fill the battery. Otherwise, if the electricity power supplied by the solar panels is not high enough for the Raspberry Pi, the battery discharges to power up the

board computer. So, the first role is to prevent the battery from overcharging if it is already full or undercharging under a threshold voltage if the battery supplies the power to the Raspberry Pi. Then, running the battery in a *partial state-of-charge* shortens its life. Thus the second role of the solar charge controller is to prevent the battery from over-discharging.

Now let us determine the characteristics of the system. On average, a Raspberry Pi 4 B with a 400% CPU load consumes a current of  $I_{max} = 1.25A$  and it is powered up with a voltage of  $V_{RPI} = 5.1V$  [44]. Hence, we can calculate the electrical power needed by the Raspberry Pi :

$$P_{RPI} = I_{RPI} * V_{RPI} = 1.25A * 5.1V = 6.375W \quad (6.1)$$

Let us consider, with some security margins, that a Raspberry Pi consumes a maximal power  $P_{RPI}^{max} = 12W$ . Now, let us determine the capacity of the battery that we need to power up the board. To do so, we considered that the battery should be able to power it up during a whole day. So, we calculated the battery capacity so that it can produce a power of  $12W$  during a period  $\Delta t_{activity} = 12h$ . In practice, as explained before, the solar charge controller prevents the battery from a complete discharge. This means that the battery never produces such a power during such a period and actually, it does not need to do so as the battery is refilled by the solar panels. However, through this assumption, we envision the worst case scenario which gives an order of magnitude for the battery capacity. From this assumption, we can calculate the energy that we want the battery to produce :

$$E_{bat} = P_{RPI}^{max} * \Delta t_{activity} = 12W * 12h = 144Wh \quad (6.2)$$

Most of the charge controllers can support  $12V$  or  $24V$  batteries. For this application, we decided to use battery of a voltage  $V_{bat} = 12V$ . With this choice of voltage, it is then possible to compute the needed capacity for the battery :

$$C_{bat} = \frac{E_{bat}}{V_{bat}} = \frac{144Wh}{12V} = 12Ah \quad (6.3)$$

Then we also need to dimension the solar panels. To do so, let us consider that the sunlight peak period during which the solar panels reach their total capacity lasts three hours. We want the solar panels to be able to refill the battery during this short time period  $\Delta t_{refill}$ . To simplify the computations, let us consider that we want to refill a battery of  $C_{bat}^{approx} = 15Ah$  during this same time period.

Let us calculate the current intensity that the solar panels should be able to produce to do so :

$$I_{solar} = \frac{C_{bat}^{approx}}{\Delta t_{refill}} = \frac{15Ah}{3h} = 5A \quad (6.4)$$

Most of the solar charge controller sold on the market support solar panels that produce a voltage of  $V_{solar} = 12V$ . From this value, we can then calculate the needed power for the solar panel to refill the battery :

$$P_{solar} = I_{solar} * V_{solar} = 5A * 12V = 60W \quad (6.5)$$

This power can be produced by disposing two solar panels of 30W in parallel.

In summary, here are the main components of the electronic circuit with their characteristics and their approximate price. The total price of the main components of the trap is 125€.

- Raspberry Pi 4 B 2GB : 5V, 1.250A max ; 30€
- Solar panels X 2 : 12V, 30W ; 60€
- Battery : 12V, 12Ah ; 35€

Note that as the trap system is autonomous and outside, these components are placed in a weatherproof box.

As the presented computations consider a worst case scenario that rarely happens in practice, the battery and the solar panels can power up the Raspberry Pi during the whole sunlight period without any interruption. However, the minimum input voltage of the solar charge controller was configured at 10.7V in order to cut the power supply of the Raspberry Pi if the battery were to be depleted.

### 6.2.2 Motion detection based algorithm

In an ideal situation on a machine with enough memory, CPU and GPU power, the trap could simply follow algorithm 2. However, the limitations associated with the use of the Raspberry Pi limit our inference time to around one minute with a default configuration. This means that the trap is only able to capture one image per minute which is not enough to sample correctly the occurrence of insects in front of the camera. It also means that the presence of an insect might not be detected if that insect shows up in front of the camera while the algorithm is in the processing phase of the previous image taken by the camera, and leaves before that phase terminates.

To avoid this issue, the first change we decided to implement was the use of a motion detection algorithm that would command the model inference. Indeed, if the inference is done only when there is a motion in front of the camera, the probability of missing an insect because the algorithm is already in the processing phase is reduced.

Our motion detection algorithm was implemented using *OpenCV* functions to first compute the absolute difference between the pixels of the image taken by

---

**Algorithm 2:** Naive real-time detection and classification algorithm.

---

**Output :** CSV file listing all detections  
**while** *True* **do**  
    Take a picture using the camera;  
    Run the inference on that image using the model;  
    Record the detections (if any) in a CSV file;  
**end**

---

the camera and the background model. This corresponds to a weighted running average of all images that were taken by the camera since the motion detector started (with a larger weight on the more recent images). After applying a threshold function over the absolute difference, we dilate in order to merge the different zones that could be created for the same motion (because of the noise). The result of these operations is used to produce contours that define the different zones where movements were perceived. The algorithm at this point only checks if there is at least one zone to start the inference on the current image, however, improvements described below make full use of them. The detailed motion detection algorithm is summarized in algorithm 3 and is inspired from the article of *Adrian Rosebrock* [45].

---

**Algorithm 3:** Motion detection algorithm.

---

**Input** : Image taken by the camera  
**Output** : A list of contours that corresponds to the zones where motion was detected  
Resize the image;  
Convert the image to grayscale;  
Blur the image;  
Add the input image to the running average with a weight *alpha*;  
Compute the pixel difference between the input image and average;  
Apply threshold on the pixel difference image;  
Dilate the thresholded image to merge zones that are close;  
Find contours of the zones in the dilated image;  
**return** Contours list

---

With *alpha*, a configuration parameter of the algorithm that controls the speed at which changes on the background are assimilated to the average background model.

### 6.2.3 Inference time analysis

While the use of a motion detector allows us to time correctly our processing with the presence of insects, it does not solve the fact that during this long inference phase, no other images are taken. However, as the precision and recall of the model are not perfect as explained in the previous chapter, it is important to have a higher level of sampling for the same insect (to see different angles, positions, etc.). Reducing the inference time is a way to increase the number of images taken for the same insect as we are able to treat more frames per second, getting us closer to real-time detection and classification.

We decided to consider the input image size as a parameter that can be modified to increase performance. We trained three different models using different input image sizes, all using the same other hyper-parameters and based on the 4 classes model described in section 5.3.2. For these models, we reported the AP and the ROC AUC metrics as well as the average inference time on Raspberry Pi 4 in figure 6.2.

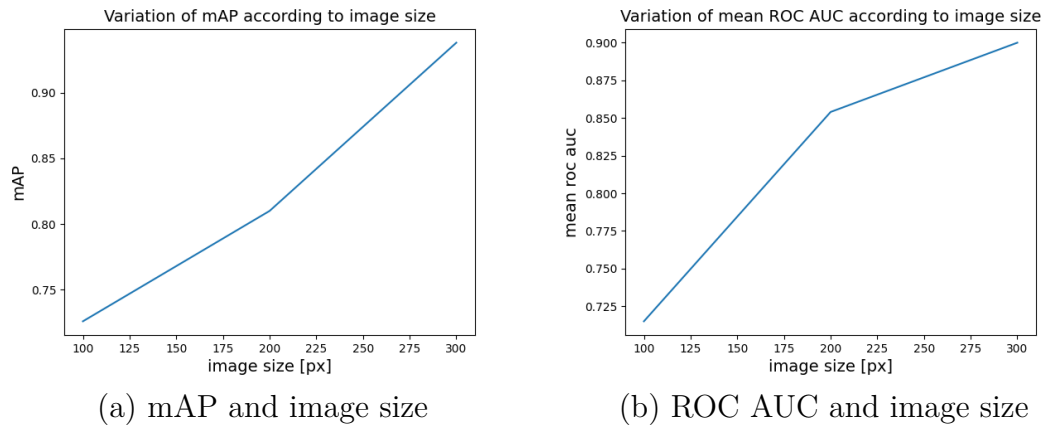


Figure 6.2: mAP, ROC AUC and inference time for different input sizes.

The three samples of these graphs are 100, 200 and 300 pixels for the width of the input image. Notice that while the ROC AUC is not reduced by much when reducing the width from 300 to 200 pixels, the mean average precision (mAP) is quite degraded. However we observe better inference times, but still not low enough to approach real-time in our current approach. Even for a 100 pixels width, the elapsed time remains around 20 seconds per image. We decided to leave the input image width to 300 in order to avoid reducing the quality of our model as the gain in inference time does not justify it.

## 6.2.4 Multiprocessing for performance

As reducing the input image's quality is not enough to drastically reduce the inference time, we decided to use multiprocessing to split the motion detection from the image processing.

The high level functional sequence diagram representing the final system running on the Raspberry Pi is presented in figure 6.3. The detection process is in charge of detecting motion and capturing the images that are fed to the inference (or processing) process. This process runs the inference for these images and saves the results. They communicate through a shared queue. The inference process first starts by initializing the model (and loading the weights), puts a *ready* flag in the queue and waits for images to be put in the queue to start the inference. This is done with blocking *gets* on the queue. The detection process first waits for the ready flag before initializing the background model. The rest is done as described above, except for the fact that the motion detector does not directly start the inference when motion is detected, but puts the image where there is potentially an insect in the shared queue for the inference process to recover. The implementation are not described in these diagrams but more information can be found in appendix B.

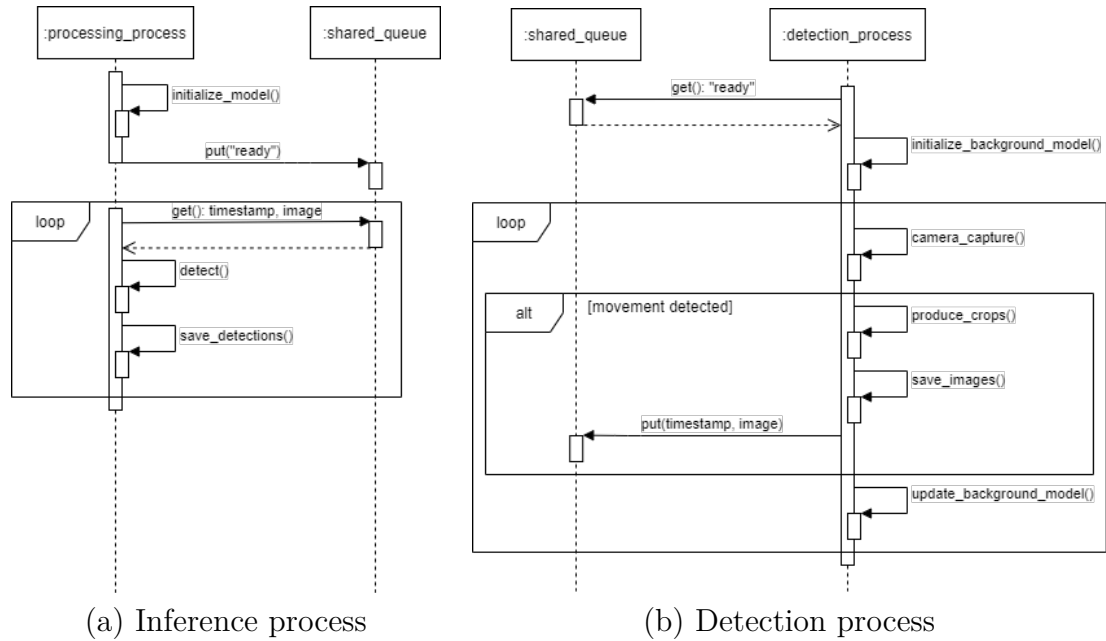


Figure 6.3: High level sequence diagrams of the different processes of the trap's system.

This implementation allows the camera to sample correctly the insect occur-

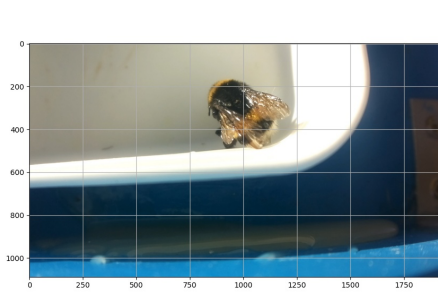
rences by capturing a greater amount of images per second, as there is no delay between a picture taken and the next one created by the inference. However, this comes at the cost of abandoning the real-time detection and classification property that didn't seem feasible with our constraints. This is not a problem in the scope of this project, the main goal being to produce an autonomous trap that allows the tracking of insects, it does not matter much if the classification and the production of bounding boxes is done a few minutes after the picture is taken. As the amount of insects passing in front of the camera is limited, the inference process can use the *idle* phase between the occurrence of two different insects to process completely the images taken of the first one.

### 6.2.5 Cropping to improve quality

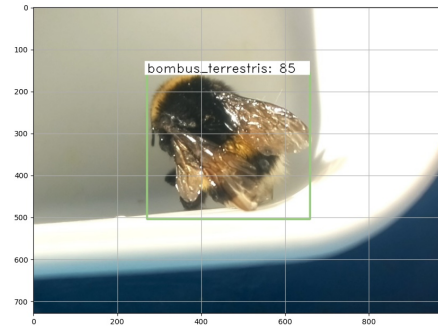
As the camera has to be placed quite far from the platform where insects land to be able to capture a greater area (and to maximize the chance of observing them), the area occupied by insects on the image is smaller than the average area taken by the insects in the training dataset. This can be explained by the fact that these images are often taken by a person and focus on the insect directly. In our case, this is not possible as we have a rigid camera that automatically captures images when detecting motion. We decided to explore the idea of cropping the images around the insects in order to increase the ratio between the image size and the insect's occupied area. We first conducted some experiments to see if the quality can be improved when we manually zoom on insects. Empirical trials with this manual zoom generally seemed to show an improvement in the quality of the detection and of the classification as shown on figure 6.4.

Observing these results, we chose to implement these crops directly in the motion detection algorithm. As mentioned in subsection 6.2.2, the motion detection algorithm used produces a set of contours between the zones where motion is detected. Instead of simply checking if there is at least one contour, we modified our detection process to produce a cropping around each of these zones with a fixed padding, and to feed them one by one in the inference process. This is represented in figure 6.3 by the functional call *produce\_crops()*. Empirical trials showed good results using this technique as presented in figure 6.5.

These changes allowed us to improve the results of the inference on images taken by the trap as they are closer to the images used for the training of the model.

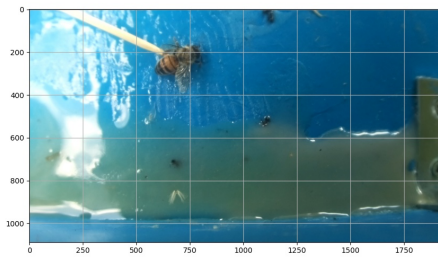


(a) Original image

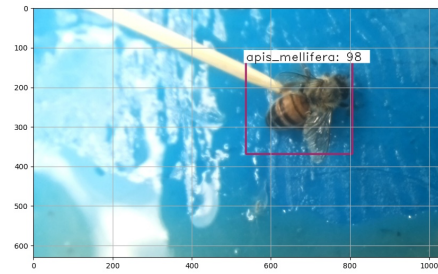


(b) Manual zoom

Figure 6.4: Detection on an image of the trap and on the manually produced crop around the insect.



(a) Original image



(b) Automatic zoom

Figure 6.5: Detection on an image of the trap and on the automatically produced crop around the insect.

## 6.3 Inference results

In section 5.4, we evaluated the model based on a dataset of pictures taken in good conditions by humans. Based on this evaluation, we determined the probability decision threshold that maximizes the F1-score. However, this threshold might be different on images produced by the trap. Indeed, we expect some differences in terms of quality of the results as the images produced by the trap are taken automatically with no human supervision and with a fixed focus. In this section, we focus on these results and try to define the decision threshold for pictures automatically taken by the trap camera. It is important to note that the experimental period took place during spring 2021. Due to an exceptionally bad weather conditions during this period, the trap could not capture a lot of insect pictures. So, the results presented in this section were produced with a small set of 81 images. These images contain 7 insects from 3 different classes only. This means that these results should be considered with a critical thinking as they maybe do not represent the real functioning of our trap system.

At first, we analyze the precision-recall curve to identify the optimal threshold that maximizes the F-score, then we quantify the results by analyzing the precision and the recall computed on the trap pictures with this threshold. Finally, in the rest of this section, we observe some specific instances in order to have a better understanding of the results given by the model.

### Optimal threshold

As we did in section 5.4 to evaluate the model on generic data, we try to find the optimal threshold that maximizes the F-score (as we do not have a preference between the precision and the recall). To do so, we can analyze the precision-recall curve plotted on figure 6.6. As expected, the quality of the model is significantly decreased compared to the results of the model on images that resemble more the training dataset.

The optimal threshold on the experimental data found is of 0.611. As it is quite similar to the threshold found with generic data, we can conclude that it is a correct choice. We can also confirm that the optimal threshold found with that generic data is applicable with experimental data. It is nevertheless slightly different than the threshold computed only on the trap data, and we could expect the model to be producing more FP due to the poor quality, justifying the need for a higher decision threshold in order to maintain a good F-score. For this reason, the rest of this section considers the results produced with the 61.1% decision threshold found with the trap data.

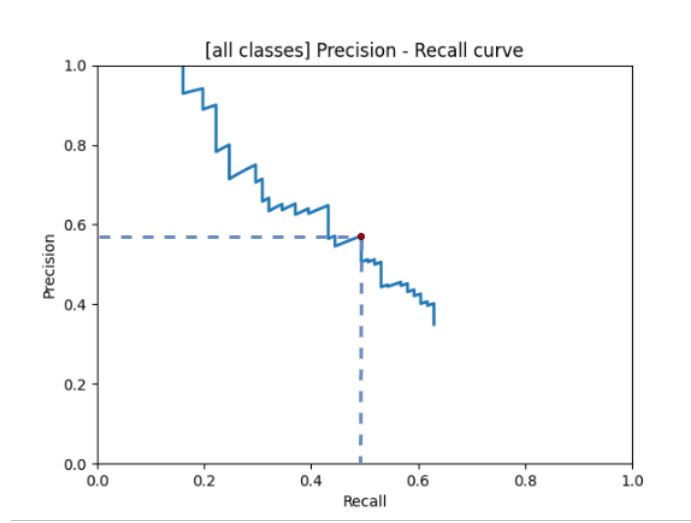


Figure 6.6: Precision-recall curve on the images of the trap.

### Evaluation of the model with experimental data

As mentioned above, the trap is not supervised by a human. Indeed, the presented application detects the insects and takes pictures of them automatically and in real-time. So, some pictures represent insects in movement and some others are not of a good quality. Thus, we do not expect the model to give results as good as the ones of the dataset that we used to theoretically evaluate the model in section 5.4. The discussion of the factors that might deteriorate the results is done in section 6.3.1.

Let us evaluate if an insect is correctly identified when it passes in front of the camera and if the predictions given by the model often correspond to the actual species of the insect that appears on the picture. To do so, we separated the pictures in different sets such that one set only contains pictures of one individual. Then on each of these sets, we compute the precision and recall metrics. The table 6.3 presents these metrics for eight different individuals.

From this table, we can also compute the means of all recall and precision values, by according the same weight to each insect. We obtain a mean recall value of 0.537. This means that when an insect passes in front of the camera, on average, it is recognized and correctly classified on a bit more than one picture out of two. The mean precision value is 0.573. This means that when the model classifies an insect, more than one prediction out of two corresponds to the correct insect species.

These results open up a prospect of improvement for a more complex insect tracking system based on our algorithm. Indeed, we can build a heuristic based

| Insect ID | Species                  | Recall | Precision | Number of pictures |
|-----------|--------------------------|--------|-----------|--------------------|
| 1         | <i>Apis mellifera</i>    | 0.625  | 0.714     | 8                  |
| 2         | <i>Apis mellifera</i>    | 0.500  | 0.555     | 20                 |
| 3         | <i>Apis mellifera</i>    | 0.467  | 0.538     | 15                 |
| 4         | <i>Bombus terrestris</i> | 0.500  | 0.400     | 4                  |
| 5         | <i>Bombus terrestris</i> | 0.333  | 0.667     | 18                 |
| 6         | <i>Vespula vulgaris</i>  | 0.583  | 0.538     | 12                 |
| 7         | <i>Vespula vulgaris</i>  | 0.750  | 0.600     | 4                  |

Table 6.1: Insect occurrences with the final version of the trap.

on the hypotheses that the mean precision of our model on all insects is over 50% and that images taken successively contain the same insect. A simple heuristic would be to attribute the label of the species that was identified the most times on a series of images to all of these images. This final decision could replace the wrong predictions with the right label. Indeed, our early results show that we have a mean precision over 50%, meaning that on average, the majority label on a series of images is the correct one. This would allow the system to reach much higher values of precision and recall as many of the FP correspond to a wrong label on an actual insect which means that also correspond to a FN (since the actual species of that insect was not identified) and that both these errors would be corrected. For 6 out of our 7 observations, that have a precision over 50%, this would be applicable and we would reach final precision and recall values closer to 1. Obviously, the boxes that were predicted where there are no insects or the insects that were not recognized at all would still contribute to a loss of precision and recall. In practice, it seems that boxes corresponding to no insects only make up 9.9% of the boxes, and the images where the insects were not recognized are on average 13.9% of the images of that insect, limiting the impact of these errors on the improvement brought by the heuristic. While the addition of that heuristic seems to be a good way to improve the model, we need to state the fact that our reasoning is based on the fact that the model seems to have a mean precision over 50% in the 7 series of images that are considered. Nevertheless, the model’s predictions directly depend on its input images and we would need a larger sample of data to be able to take these conclusions.

In the case of our insect 4, the precision is under 50%, meaning that the majority label is not especially the correct one. The images corresponding to that insect are shown in figure 6.7. We can notice that the incorrect labels of *bombus\_hortorum* are predicted as many times as the correct *bombus\_terrestris* labels. However, the probability associated to the correct labels is higher than the one with the

incorrect labels. Indeed, in a series of images of the same insect, we will observe that on average, the correct predictions have higher associated probabilities than the incorrect predictions. With this in mind, we could include this probability to the heuristic in order to improve the final choice and to increase the confidence in its results. This way, we would be allowing the cases where the actual label is not the majority one to possibly be classified correctly.

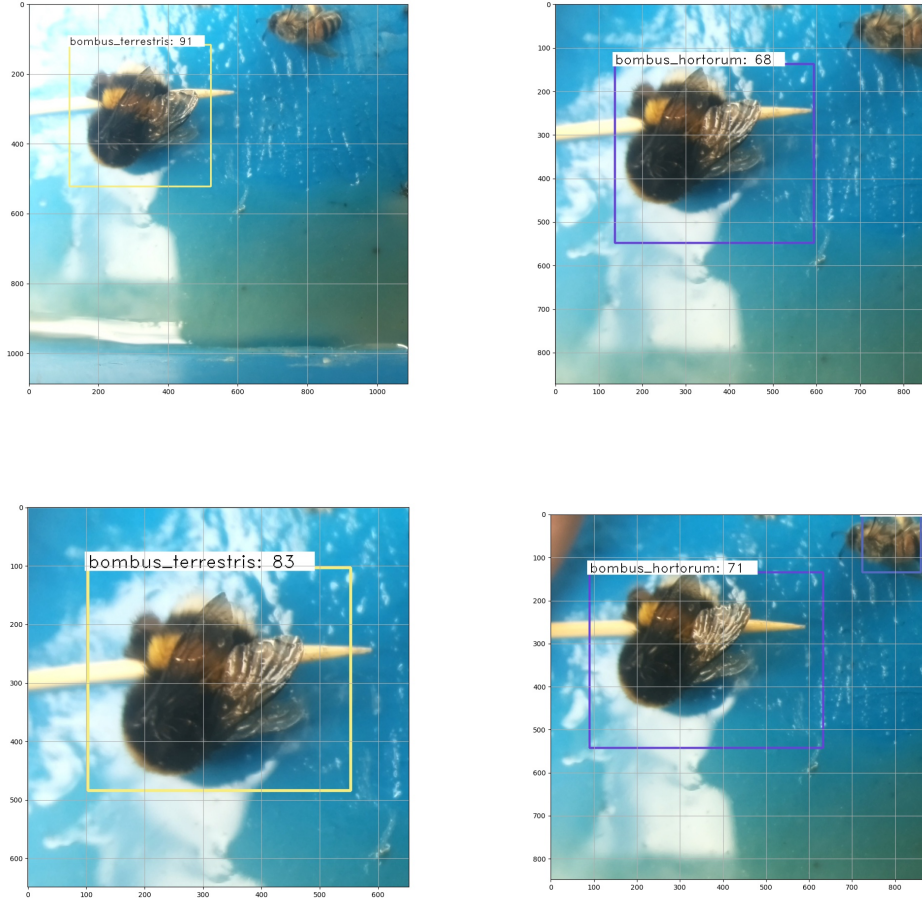
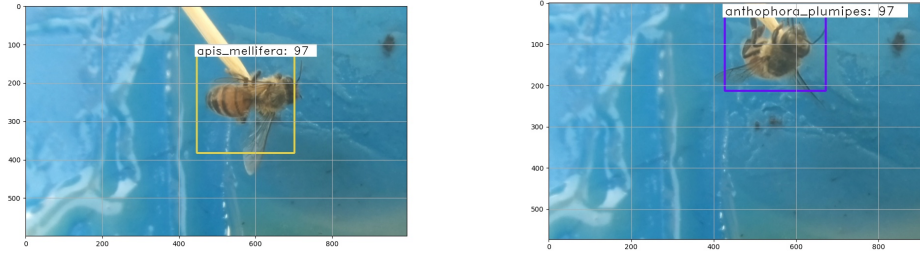


Figure 6.7: Comparison of results for insect ID = 4 (*Bombus terrestris*).

Finally, it is also important to mention that there is no recall value of 0% in table 6.3. This would correspond to a situation where an insect would not be recognized on any of the pictures taken. So, we can deduce that all the insects are correctly identified by the model on at least one picture. This means that an application that aims at detecting the presence of a specimen that is part of our



(a) Correct prediction

(b) Erroneous prediction

Figure 6.8: Comparison of results for insect ID = 2 [42].

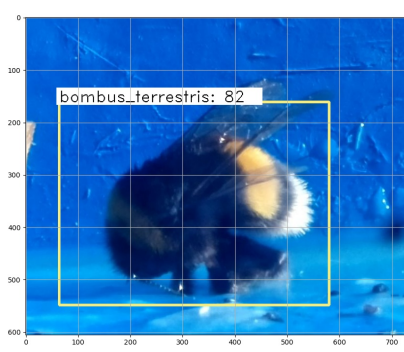
species of interest would entirely be compatible with our solution.

### 6.3.1 Qualitative analysis

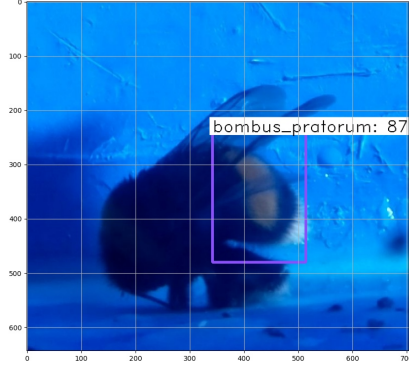
In this subsection, we interpret visual results obtained for some insects presented in table 6.3.

As we can observe in figure 6.8, the insect posture influences the prediction of the model. Indeed, when the picture corresponds to a top view of the insect, the model correctly predicts the species but when the picture corresponds to a front view of the insect, this one could be misclassified. This can be explained by the fact that the training dataset mostly contains pictures with top view of insects. Moreover, it is possible that the distinctive characteristics that help for an insect identification may not appear when the insect is not in an optimal posture. Taking the time to compose a more diverse dataset using professional as well as experimental images (images that are not taken in perfect conditions) could ease the inference for the model in those special cases.

The performance of the model can also be influenced by the luminosity of the picture taken by the Raspberry Pi camera. Indeed, as we can observe on figure 6.9, if the surrounding luminosity decreases, for example due to a cloudy weather, the camera could not precisely distinguish the colours of an individual. It is therefore difficult for the model to correctly detect and classify it. Adding a light to the trap that homogenizes the luminosity, or using preprocessing methods to standardize the images and reduce the impact of the surroundings could be good candidates to



(a) Correct prediction



(b) Erroneous prediction

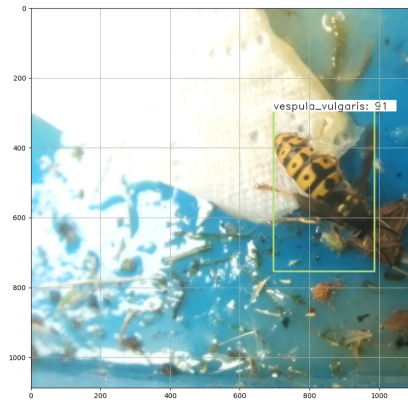
Figure 6.9: Comparison of results for insect ID = 5 [42].

avoid those mistakes.

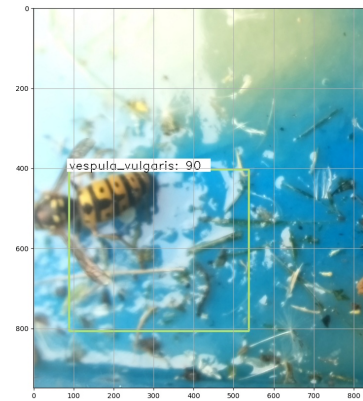
It is also possible that the model correctly classifies an observed insect but produces a bounding box that does not precisely matches the insect. This situation is illustrated in figure 6.10. In the right picture of this image, the bounding box that is supposed to show the *Vespula vulgaris* is too large. A possible hypothesis could be the fact that model confuses the blades of grass with the black stripes present on the abdomen of the wasp. In general, the background of the trap may be more messy than the backgrounds of the training images which would contribute in reducing the quality of the detections. Moreover, as mentioned in subsection 5.2.2, the boxes in the training dataset were created using a simplified model trained with the Mask R-CNN algorithm. This model could produce some inaccurate boxes. This could also justify the imprecision of some bounding boxes predictions by our final model. Redefining manually all the bounding boxes used for the training of the model in order to reduce that imprecision could help improve the final results.

Sometimes, the RPN produces two different boxes for the same region of interest, as represented in figure 6.11. It is then possible that the classifier attributes two different labels on these two boxes. This situation could be avoided by modifying the IoU value above which the RPN decides to merge two overlapping boxes. However, by varying this threshold, it is possible that the model merges two boxes of distinct specimens too. As in practice, this situation does not often happen, it does not represent an important problem. This only slightly decreases the prediction value of the model.

In short, several factors could decrease the precision and the recall of the model

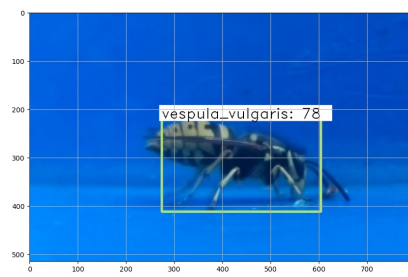


(a) Correct prediction

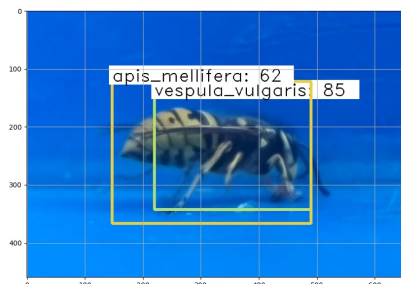


(b) Erroneous prediction

Figure 6.10: Comparison of results for insect ID = 6 [42].



(a) Correct prediction



(b) Erroneous prediction

Figure 6.11: Comparison of results for insect ID = 7 [42].

on experimental images. Among these factors, we can namely cite the insect posture, the image luminosity, the image quality, the trap position or the picture background. More generally, as expected, the model performs worst on pictures that do not resemble the pictures contained in the training dataset. Some solutions to reduce the impact of these factors are mentioned and could help avoid some of the mistakes made by the model. These are further described in section 7.1.

## 6.4 Conclusion

For the physical part of this thesis' project, we built an insect trap that attracts the insects in front of the camera by using honey as a bait. This trap is autonomous as it operates the whole day by the use of solar energy.

As the detection time is long with Faster R-CNN algorithm, we used multiprocessing to reduce the impact of that delay. There are two different processes running on the Raspberry Pi. The first one, which is not CPU consuming, takes pictures with the Raspberry Pi's camera when a movement is detected. The second one aims at processing the images through the trained network without impacting the movement detection.

To make the taken pictures resemble the dataset used to train the model, we crop the pictures around the movement area, so that the ratio of the box area on the picture area is increased. We evaluated the model based on a few number of individuals, which means that the obtained results could be different if the experiment phase takes place during a larger period. However, we can expect the model to perform in the final trap environment with an expected precision and an expected recall per species respectively of 0.537 and 0.573. These low metrics values could be explained in practice by the fact that there is a quite significant difference between the images in the training dataset and the images automatically taken by the Raspberry Pi camera. We could enhance the model performance by developing a heuristic that considers the images taken in a defined time interval and observes their predominant label as well as the best prediction probability values. The benefits and disadvantages of a possible heuristic are discussed in section 7.1.

# Chapter 7

## Conclusion

Tracking and identifying insects can be a very tedious and time-consuming task. Nowadays it is performed by professional entomologists who employ diverse methods like the use of determination keys.

In this thesis, we tried to prove that the tracking of insects can be automated with the use of machine learning. To do so, we imagined a prototypical system that focuses on ten different *Hymenoptera* species, that belong to *Apis*, *Bombus*, *Anthophora* and *Vespula* genus. Concretely, we analyzed the performance of a prediction model deployed on a physical trap application.

In this thesis, we started our development with a summary of some fundamental concepts of machine learning needed to understand our whole solution in part I. Afterwards we continued our development with a review of the existing object detection architectures that could be used in our solution and a discussion about some insect detection projects involving machine learning which are similar to ours in part II. This allowed us to select two candidate architecture for our project, Faster R-CNN and Mask R-CNN, that both solve the object detection problem with different requirements in terms of input data.

Finally, we ended up the report with our own solution in part III. In this part, we stated the goals and constraints of the final application and described our whole methodology to achieve the goal of the thesis, namely the way we trained our model with a validation procedure and the way we assessed its performance. We chose Faster R-CNN as our object detection architecture to respect the stated constraints, namely the fact that the model needs to be deployed on the Raspberry Pi, which presents limited performance. We also described how we imagined the physical insect trap to attract the *Hymenoptera* species of interest and take high quality pictures of them, by exploring the multi-processing and image cropping mechanisms. This part also aimed at presenting experimental results the model can achieve with both professional and automatic pictures.

To solve the exposed problem, we proposed a complete solution that consists of

an object detection model trained using around 15000 images, which is deployed on a physical trap. As stated above, we chose Faster R-CNN as it seems to outperform other detection algorithms like Fast R-CNN or R-CNN. Its alternative, Mask R-CNN, which is able to solve the semantic segmentation problem, was not selected in our final solution as for the application of insect tracking, the production of bounding boxes is sufficient to localize the individuals, and its requirements on input data are too heavy for our project.

During the training phase of the model, we applied several augmentation techniques that enhance its performance. Some of them are classically used in this algorithm, like image rotation and image flipping, but we also noticed that applying a brightness jitter technique on the image can decrease the total loss of the network.

Finally, we also designed an autonomous insect trap built up with a Raspberry Pi on which the algorithm is deployed and a high quality camera. This system can be powered up during a whole day by the use of solar energy and a charge controller. To attract the species of interest in front of the camera lens, we built up a blue platform in front of it on which we disposed some honey as a bait.

We can analyze the results achieved by our solution in two different ways. The first way consists in evaluating the performance of the model in favorable conditions, which means by testing it on an images testing set very close to the training dataset, composed of pictures taken by human beings. We concluded that the model performs with an AP score of 73.9% and a ROC AUC value of 71.9%. Moreover, the theoretical optimal detection probability threshold that maximizes the F1-score is 60%. This detection threshold leads the model to perform with a recall value of 74.1% and a precision value of 76.8%. This corresponds to a F1-score of 0.754.

The second way of evaluation consists in observing the performance of the model based on some pictures automatically taken by the trap in its final conditions. To do so, we computed the recall and precision scores for each individual that passed in front of the camera. We conclude that the mean precision score and the mean recall score over all the specimen are respectively 57.3% and 53.7%. This means that an insect is correctly identified on more than one of two pictures. Moreover, we noticed that an insect is always detected on at least one of its corresponding pictures. Note that due to weather constraints, these results were produced with a few number of individuals.

To conclude, in this work, we proved that building an autonomous application that automates the tracking of insects is entirely feasible. Such an application can be useful in the field of entomology research, namely for census and observations tasks. However, the presented performance of the final application system should be enhanced in several different ways. Some possible improvements are described

in the next section.

## 7.1 Further work

There are two main points of possible improvement on our final system. The first point is the model training that can be improved to increase the quality of inference. The second point consists in modifying the trap system to reason differently on the results of the model or to harvest data of a higher quality.

### Improving the model

Firstly, and as mentioned in chapter 5, we made some compromises that reduce the quality of the model to improve the performance at training and inference time. With a better hardware for the training and for the trap, these compromises would not have to be made anymore and we could, for instance, use a wider backbone, or have larger input images than the limiting 300x300 pixels size used. Therefore, both the quality and the performance at inference time of the model could be greatly improved. We can also use other image preprocessing techniques that were not considered in the scope of this project to increase the quality of the final results. The training dataset could also be improved as we are currently using generic data that has not been produced by the trap. Indeed, without the time constraint of our project, we could have easily imagined a harvesting phase that would have used a first version of the trap (that simply takes pictures when detecting motion) to capture a great amount of images that could have been added to the training dataset. This could greatly increase the performance of the inference on experimental images (from the trap) as the dataset would resemble more this kind of images. Generally, increasing the size of the dataset also improves the model as we have a stronger representation of our classes. Adding another class composed of *outliers* is also a possible improvement. Indeed, for now, insects (or objects) that share similar characteristics with the different species on which the model was trained can be possibly detected as one of these species, introducing some FP in the detections. This additional class could reduce the amount of outliers and possibly increase the precision of the model, though it might also increase the confusion between the existing classes and this new class. The annotation system is also imperfect as mentioned in section 6.3.1, as the bounding boxes of the training set are produced automatically, we observe imperfect results at inference time. This is why a manual annotation process (that could not have fit within the time constraint of this project) or a more advanced annotation system could improve the quality of the detections.

## Improving the trap system

Secondly, our model could be used as a basis to make more accurate decisions, such as in the heuristic presented in section 6.3. The results presented seem to imply that a heuristic that is based on the fact that detections on consecutive images correspond to the same insect, and that the predictions that are done the most on these series of images and with the highest associated probability can effectively and greatly improve the precision and the recall of the system. However, this kind of heuristic faces some challenges that could be subject of further research. Indeed, whereas it might work correctly in the simple case where all consecutive images contain the same insect, this can possibly not always be the case in practice. It is possible that several insects could be present on the same images and that this heuristic would have to be able to consider each of these insects individually. Other model architecture can also use this time information to learn the motion of objects and track the insect on the image to be able to handle these situations better [46]. Another area for improvement would be the trap itself. Indeed, there might be better solutions to attract more insects, or to improve the quality of the images taken, by adding a light that would homogenize the luminosity of the pictures taken by the camera and reduce the impact of clouds or shadows.

In short, our project already produced satisfactory results that can already be the basis for effective detection systems to improve the scientific research field, namely in the sensing or observation procedures. However, it still leaves some areas open for improvement and can possibly be greatly enhanced.

# Bibliography

- [1] *Benefits of Insects / Science Literacy and Outreach / Nebraska*. entomology.unl.edu. URL: <https://entomology.unl.edu/scilit/benefits-insects#:~:text=Insects%5C%20provide%5C%20useful%5C%20services%5C%20to> (visited on 06/09/2021).
- [2] *Hymenoptera - ants, bees, wasps*. www.ento.csiro.au. URL: <https://www.ento.csiro.au/education/insects/hymenoptera.html>.
- [3] John A. Lee. *LELEC2870 - Machine learning : regression, deep networks and dimensionality reduction - A (not so) quick overview of Deep Learning*. 2020.
- [4] Ross Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: *Thecvf.com* (2014), pp. 580–587. URL: [https://openaccess.thecvf.com/content\\_cvpr\\_2014/html/Girshick\\_Rich\\_Feature\\_Hierarchies\\_2014\\_CVPR\\_paper.html](https://openaccess.thecvf.com/content_cvpr_2014/html/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.html) (visited on 04/13/2021).
- [5] Nirav Bhatt and Deven J Patel. *Insect Identification Among Deep Learning’s Meta-architectures Using TensorFlow*. ResearchGate, Oct. 2019. URL: [https://www.researchgate.net/publication/337917435\\_Insect\\_Identification\\_Among\\_Deep\\_Learning%5C%27s\\_Meta-architectures\\_Using\\_TensorFlow](https://www.researchgate.net/publication/337917435_Insect_Identification_Among_Deep_Learning%5C%27s_Meta-architectures_Using_TensorFlow) (visited on 10/04/2020).
- [6] Dan Jeric et al. *A Real-time Multi-class Insect Pest Identification Method using Cascaded Convolutional Neural Networks*. ResearchGate, May 2018. URL: [https://www.researchgate.net/publication/333916287\\_A\\_Real-time\\_Multi-class\\_Insect\\_Pest\\_Identification\\_Method\\_using\\_Cascaded\\_Convolutional\\_Neural\\_Networks](https://www.researchgate.net/publication/333916287_A_Real-time_Multi-class_Insect_Pest_Identification_Method_using_Cascaded_Convolutional_Neural_Networks) (visited on 10/04/2020).
- [7] Qing Yao et al. “An Insect Imaging System to Automate Rice Light-Trap Pest Identification”. In: *Journal of Integrative Agriculture* 11 (June 2012), pp. 978–985. DOI: [10.1016/S2095-3119\(12\)60089-6](https://doi.org/10.1016/S2095-3119(12)60089-6). URL: <https://www.sciencedirect.com/science/article/pii/S2095311912600896> (visited on 10/04/2020).

- [8] *Raspberry Pi 4 Model B specifications – Raspberry Pi*. Raspberrypi.org, 2019. URL: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>.
- [9] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (June 2017), pp. 1137–1149. DOI: [10.1109/tpami.2016.2577031](https://doi.org/10.1109/tpami.2016.2577031).
- [10] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Medium, Dec. 2018. URL: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53> (visited on 10/13/2020).
- [11] Asifullah Khan et al. “A survey of the recent architectures of deep convolutional neural networks”. In: *Artificial Intelligence Review* 53 (Apr. 2020), pp. 5455–5516. DOI: [10.1007/s10462-020-09825-6](https://doi.org/10.1007/s10462-020-09825-6). URL: <https://link.springer.com/article/10.1007/s10462-020-09825-6> (visited on 10/11/2020).
- [12] Shashi Sathyanarayana. *A Gentle Introduction to Backpropagation*. ResearchGate, July 2014. URL: [https://www.researchgate.net/publication/266396438\\_A\\_Gentle\\_Introduction\\_to\\_Backpropagation](https://www.researchgate.net/publication/266396438_A_Gentle_Introduction_to_Backpropagation) (visited on 06/11/2021).
- [13] Computer Science Wiki. *Max-pooling / Pooling*. 2018. URL: [https://computer-science-wiki.org/index.php?title=Max-pooling/\\_Pooling%5C&oldid=7839](https://computer-science-wiki.org/index.php?title=Max-pooling/_Pooling%5C&oldid=7839) (visited on 04/13/2021).
- [14] Raúl Gómez. *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. Github.io, 2018. URL: [https://gombru.github.io/2018/05/23/cross\\_entropy\\_loss/](https://gombru.github.io/2018/05/23/cross_entropy_loss/).
- [15] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. Sebastian Ruder, Jan. 2016. URL: <https://ruder.io/optimizing-gradient-descent/index.html#adam> (visited on 04/08/2021).
- [16] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. Machine Learning Mastery, Apr. 2019. URL: <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>.
- [17] Muneeb ul Hassan. *VGG16 - Convolutional Network for Classification and Detection*. Neurohive.io, Nov. 2018. URL: <https://neurohive.io/en/popular-networks/vgg16/>.

- [18] Connor Shorten. *Introduction to ResNets*. Medium, Jan. 2019. URL: <https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4>.
- [19] Shervin Minaee. *20 Popular Machine Learning Metrics. Part 1: Classification & Regression Evaluation Metrics*. Medium, Oct. 2019. URL: <https://towardsdatascience.com/20-popular-machine-learning-metrics-part-1-classification-regression-evaluation-metrics-1ca3e282a2ce>.
- [20] *Precision-Recall*. Scikit-learn. URL: [https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_precision\\_recall.html#sphx-gl-r-auto-examples-model-selection-plot-precision-recall-py](https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html#sphx-gl-r-auto-examples-model-selection-plot-precision-recall-py) (visited on 05/08/2021).
- [21] Francis Chulu et al. *A Convolutional Neural Network for Automatic Identification and Classification of Fall Army Worm Moth*. ResearchGate, Aug. 2019. URL: [https://www.researchgate.net/publication/334824144\\_A\\_Convolutional\\_Neural\\_Network\\_for\\_Automatic\\_Identification\\_and\\_Classification\\_of\\_Fall\\_Army\\_Worm\\_Moth](https://www.researchgate.net/publication/334824144_A_Convolutional_Neural_Network_for_Automatic_Identification_and_Classification_of_Fall_Army_Worm_Moth) (visited on 10/04/2020).
- [22] *OpenCV library*. Opencv.org, 2019. URL: <https://opencv.org/>.
- [23] *scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation*. Scikit-learn.org, 2019. URL: <https://scikit-learn.org/stable/>.
- [24] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. “Efficient Graph-Based Image Segmentation”. In: *International Journal of Computer Vision* 59 (Sept. 2004), pp. 167–181. DOI: [10.1023/b:visi.0000022288.19776.77](https://doi.org/10.1023/b:visi.0000022288.19776.77). URL: <https://link.springer.com/article/10.1023/B:VISI.0000022288.19776.77> (visited on 04/13/2021).
- [25] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine Learning* 20 (Sept. 1995), pp. 273–297. DOI: [10.1007/bf00994018](https://doi.org/10.1007/bf00994018). URL: <https://link.springer.com/article/10.1007/BF00994018> (visited on 06/11/2021).
- [26] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. Medium, July 2018. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e#:~:text=The%5C%20approach%5C%20is%5C%20similar%5C%20to%5C%20the%5C%20R%5C%2DCNN%5C%20algorithm.%5C&text=The%5C%20reason%5C%20%5C%20E2%5C%80%5C%9CFast%5C%20R%5C%2DCNN,map%5C%20is%5C%20generated%5C%20from%5C%20it>. (visited on 04/14/2021).
- [27] Ross Girshick. *Fast R-CNN*. arXiv.org, 2015. URL: <https://arxiv.org/abs/1504.08083> (visited on 04/14/2021).

- [28] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. arXiv.org, 2015. URL: <https://arxiv.org/abs/1506.01497> (visited on 04/13/2021).
- [29] Mira Park et al. *Marine Vertebrate Predator Detection and Recognition in Underwater Videos by Region Convolutional Neural...* ResearchGate, Aug. 2019. URL: [https://www.researchgate.net/publication/335360649\\_Marine\\_Vertebrate\\_Predator\\_Detection\\_and\\_Recognition\\_in\\_Underwater\\_Videos\\_by\\_Region\\_Convolutional\\_Neural\\_Network](https://www.researchgate.net/publication/335360649_Marine_Vertebrate_Predator_Detection_and_Recognition_in_Underwater_Videos_by_Region_Convolutional_Neural_Network) (visited on 04/18/2021).
- [30] Kaiming He et al. *Mask R-CNN*. Jan. 2018. URL: <https://arxiv.org/pdf/1703.06870v3.pdf?fbclid=IwAR3txzGUQbK5lFUBRrI9MoH4LislgC1B71WntKlPng5XzICwvVuWWzbucY0> (visited on 11/09/2020).
- [31] *Observations.be, Natagora, Natuurpunt et la Fondation "Observation International"*. Observations.be. URL: <https://observations.be>.
- [32] *GBIF*. Gbif.org, 2021. URL: <https://www.gbif.org/>.
- [33] Abhishek Dutta. *VGG Image Annotator*. Ox.ac.uk, 2012. URL: <https://www.robots.ox.ac.uk/~vgg/software/via/via.html> (visited on 05/04/2021).
- [34] Vahid Beyraghi and Jonathan Thibaut. *vahidbe : Master\_Thesis - Bees dataset and models*. GitHub, May 2021. URL: [https://github.com/vahidbe/Master%5C\\_Thesis%5C\\_Bees%5C\\_Dataset%5C\\_and%5C\\_Models](https://github.com/vahidbe/Master%5C_Thesis%5C_Bees%5C_Dataset%5C_and%5C_Models) (visited on 06/03/2021).
- [35] leekunhee. *leekunhee/Mask\_RCNN*. GitHub, May 2021. URL: [https://github.com/leekunhee/Mask\\_RCNN](https://github.com/leekunhee/Mask_RCNN) (visited on 06/03/2021).
- [36] RockyXu66. *RockyXu66/Faster\_RCNN\_for\_Open\_Images\_Dataset\_Keras*. GitHub, May 2021. URL: [https://github.com/RockyXu66/Faster%5C\\_RCNN%5C\\_for%5C\\_Open%5C\\_Images%5C\\_Dataset%5C\\_Keras](https://github.com/RockyXu66/Faster%5C_RCNN%5C_for%5C_Open%5C_Images%5C_Dataset%5C_Keras) (visited on 06/03/2021).
- [37] Pascal Getreuer. "A Survey of Gaussian Convolution Algorithms". In: *Image Processing On Line* 3 (Dec. 2013), pp. 286–310. DOI: [10.5201/ipol.2013.87](https://doi.org/10.5201/ipol.2013.87). URL: [https://www.ipol.im/pub/art/2013/87/?utm\\_source=doi](https://www.ipol.im/pub/art/2013/87/?utm_source=doi) (visited on 06/11/2021).
- [38] Shalhav Zohar. "The Solution of a Toeplitz Set of Linear Equations". In: *Journal of the ACM* 21 (Apr. 1974), pp. 272–276. DOI: [10.1145/321812.321822](https://doi.org/10.1145/321812.321822). URL: <https://web.njit.edu/~akansu/PAPERS/Haddad-AkansuFastGaussianBinomialFiltersIEEE-TSP-March1991.pdf> (visited on 06/11/2021).

- [39] P.E. Trahanias and A.N. Venetsanopoulos. “Color image enhancement through 3-D histogram equalization”. In: *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol. IV. Conference D: Architectures for Vision and Pattern Recognition*, (2021). DOI: [10.1109/icpr.1992.202045](https://doi.org/10.1109/icpr.1992.202045). URL: <https://ieeexplore.ieee.org/document/202045/?arnumber=202045> (visited on 06/11/2021).
- [40] *Luminance*. Schorsch.com, 2020. URL: <https://www.schorsch.com/en/kbase/glossary/luminance.html> (visited on 06/11/2021).
- [41] Alvy Ray Smith. *Gamma Correction*. Sept. 1995. URL: <https://www.cs.princeton.edu/courses/archive/fall00/cs426/papers/smith95d.pdf> (visited on 06/11/2021).
- [42] *Understanding Gamma Correction*. www.cambridgeincolour.com. URL: <https://www.cambridgeincolour.com/tutorials/gamma-correction.htm>.
- [43] *Bombus terrestris (Linnaeus, 1758) - Bourdon terrestre (Le)*. Inventaire National du Patrimoine Naturel, 2014. URL: [https://inpn.mnhn.fr/espece/cd\\_nom/53104/tab/fiche](https://inpn.mnhn.fr/espece/cd_nom/53104/tab/fiche) (visited on 06/11/2021).
- [44] Jeff Geerling. *Power Consumption Benchmarks | Raspberry Pi Dramble*. Pidramble.com, 2019. URL: <https://www.pidramble.com/wiki/benchmarks/power-consumption>.
- [45] Adrian Rosebrock. *Basic motion detection and tracking with Python and OpenCV - PyImageSearch*. PyImageSearch, May 2015. URL: <https://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and-tracking-with-python-and-opencv/> (visited on 04/06/2021).
- [46] *Object tracking: A survey: ACM Computing Surveys: Vol 38, No 4*. ACM Computing Surveys (CSUR), 2017. URL: <https://dl.acm.org/doi/abs/10.1145/1177352.1177355> (visited on 05/17/2021).
- [47] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.47725z> (visited on 03/22/2021).
- [48] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.rbnwb5> (visited on 03/29/2021).
- [49] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.ep4rp5> (visited on 03/29/2021).
- [50] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.kjdygy> (visited on 03/29/2021).
- [51] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.4m8byt> (visited on 03/27/2021).

- [52] *GBIF Occurrence Download*. GBIF.org. URL: <https://doi.org/10.15468/dl.txuckp> (visited on 05/02/2021).
- [53] Anonymous. *Introduction to entomology*. URL: <http://www.missouricareereducation.org/doc/entomology/EntomologySR.pdf>.
- [54] *Glossary of Terms*. www.ento.csiro.au. URL: <https://www.ento.csiro.au/education/glossary.html> (visited on 04/15/2021).
- [55] George Hammond and Madison Blankenship. *Apis mellifera (honey bee)*. Animal Diversity Web, 2009. URL: [https://animaldiversity.org/accounts/Apis\\_mellifera/](https://animaldiversity.org/accounts/Apis_mellifera/).
- [56] *Anthophora plumipes (Pallas,1772) / BWARS*. www.bwars.com. URL: <https://www.bwars.com/index.php?q=bee/apidae/anthophora-plumipes> (visited on 06/11/2021).
- [57] Ted Benton. *Bumblebees*. Collins, 2006, pp. 351–355.
- [58] Ted Benton. *Bumblebees*. Collins, 2006, pp. 283–292.
- [59] Ola Jennersten and Sven G. Nilsson. “Insect Flower Visitation Frequency and Seed Production in Relation to Patch Size of *Viscaria vulgaris* (Caryophyllaceae)”. In: *Oikos* 68 (Nov. 1993), p. 283. DOI: [10.2307/3544841](https://doi.org/10.2307/3544841). (Visited on 03/25/2019).
- [60] *Bombus lucorum – The White Tailed Bumblebee*. BuzzAboutBees.net. URL: <https://www.buzzaboutbees.net/bombus-lucorum.html> (visited on 06/11/2021).
- [61] *Common carder bumblebee - Encyclopedia of Life*. eol.org. URL: <https://eol.org/pages/1065209>.
- [62] Ted Benton. *Bumblebees*. Collins, 2006, pp. 338–342.

# Appendices

# Appendix A

## Basic concepts of entomology

This appendix aims to propose to the reader a more detailed overview of the species that we focus on in the scope of this project. It briefly describes the basic entomology concepts that rule the insect classification, and mentions the main distinguishing characteristics of the insects considered by our system.

### A.1 Insect identification

The insect identification is mainly done by observing their characteristics and by classifying them namely according to their family, genus or species.

#### A.1.1 Distinguishing characteristics of an insect

The classical insect identification techniques consist in observing their three main body sections, namely head, thorax and abdomen (see Figure A.1).

Insects' heads can be distinguished by several characteristics. Eyes are either compound or single. Compound eyes are large eyes found on most adult insects while simple eyes are small eyes located on top of the head of adults. Insects' heads are also composed of antennae, which have sensors to detect the sound, taste, odor and feel the surrounding environment. These antennae can be of various shape that are proper to some insect species. The mouth part can also be distinguished according to chewing insects like beetles for example or piercing-sucking insects like mosquitoes (see Figure A.2 - (a)).

The thorax is the second part of the insects and contains muscles. This part is thus responsible for the insect movements. The insect's legs are attached to this part. Each leg has five parts that are differently arranged from one insect to the other according to the leg function, such as jumping, running, grasping or swimming (see Figure A.2 - (b)). The insect's wings are also attached to this body

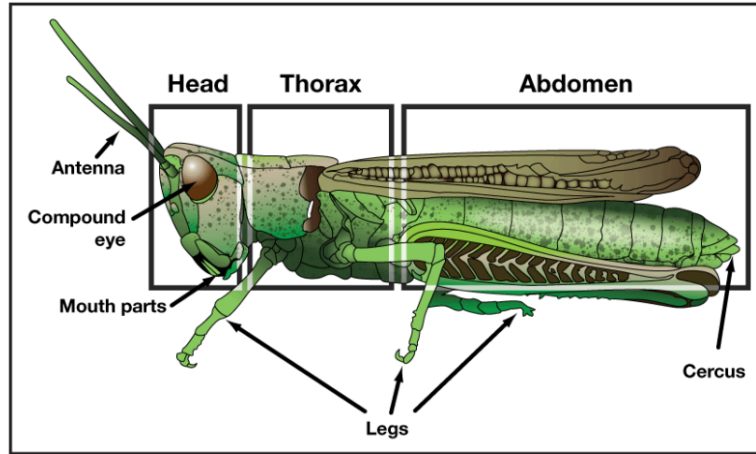


Figure A.1: Three main insect sections [53].

part. These also involve a great variety in shape, color, size, vein pattern and thickness according to the different species.

The abdomen is the third and last part of the insect body. This part may be visible or underneath the wings. This section contains the internal organs of the insect and the sexual organs. Depending on the species, this section may also have a needle-like projection for piercing or stinging [53].

### A.1.2 Insects classification

The animal kingdom is divided into different groups and subgroups. Insects belong to one of the basic divisions of this kingdom called *Arthropoda*. This is divided into several classes, namely the *Insecta* class. This class is further divided into several orders that can be distinguished from different characteristics that are partially mentioned in section A.1.1. Orders are then divided into groups. Each of these groups become more and more specific until, at last, each insect species can be identified by its own name.

*Insecta* class is very wide and contains a lot of species. Thus, there exists some books such as field guides or insect keys that are used to precisely identify insect species. These books contain characteristics description of insects, information about their lifestyle, their environment and how the insects are related to each other. There are 29 orders in total in *Insecta* class. The species tracked in the presented application and classified by the CNN all belong to the *Hymenoptera* order [53].

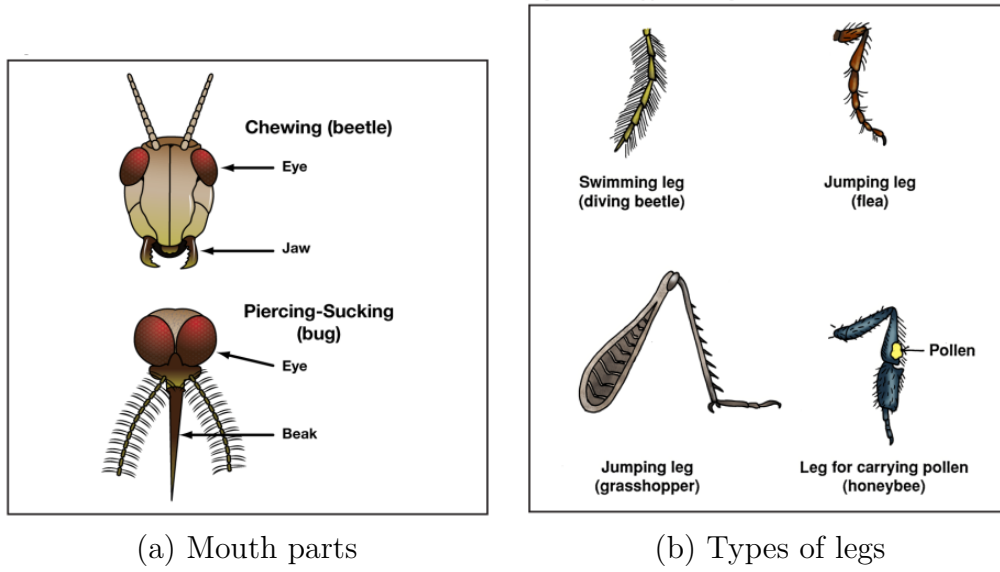


Figure A.2: Insect anatomy characteristics [53].

## A.2 Hymenoptera

The *Hymenoptera* order is one of the most beneficial insect orders to humans. Lots of species that belong to this order are pollinators of wild and cultivated flowering plants. *Hymenoptera* is an order from *Insecta* class, which belongs to *Arthropoda* phylum in *Animalia* kingdom. *Hymenoptera* namely contains species of ants, bees or wasps.

*Hymenopterans* can be recognised by their particular characteristics. They usually have two pairs of membranous wings. The forewings are larger than the hindwings and are held together by small hooks as represented in figure A.3. Females usually have a hardened ovipositor<sup>1</sup>. Many of them have a constriction between the first 2 segments of the abdomen, which is known as 'wasp waist'. Finally *hymenopterans* also usually have compound large eyes and chewing mandibulate mouthparts, although in some species the lower lip is modified to form a tongue [2].

This work focuses on ten different *Hymenoptera* species. These ten species of interest are briefly described in the following sections. Individuals pictures are presented in figures A.4 and A.5.

<sup>1</sup>Ovipositor: Tubular apparatus, usually concealed but sometimes extending some length outside the end of the abdomen, with which many female insects deposit eggs [54].



Figure A.3: Wing hooks of a wasp.

## ***Apis mellifera***

*Apis mellifera*, more classically called honey bees, are red/brown with black bands and orange yellow rings on abdomen. Their thorax is more hairy than their abdomen. Their legs are dark brown/black and they can be recognized by the pollen basket on their hind legs. The queen size varies from 15 millimeters to 20 millimeters while the workers bees grow up to 11-13 millimeters [55].

## ***Anthophora plumipes***

*Anthophora plumipes* grow up to 13-15 millimeters. Their body is always hairy but this species shows a sexual dimorphism. Males have bright redish-brown hair or gray hair, while females are usually black or dark brown and have reddish orange hairs on the hind tibia. Males also distinguish from females by having very elongated middle legs and, long hair on its mid tarsi<sup>2</sup> and the lower face yellow or cream-coloured, rather than black [56].

## ***Bombus hortorum***

*Bombus hortorum* have an oval head and a tongue that is about 15 mm long. Their body can have a length from 13 millimeters to 20 millimeters, varying from males, workers or queens. Their color is black with a yellow collar, a narrow yellow band on the scutellum, and a third yellow band on the first and second terga<sup>3</sup>. The tail is white [57].

---

<sup>2</sup>Tarsus: Final segment of the leg of the insect.

<sup>3</sup>Tergum: Thickened dorsal plate on each segment of the body of an arthropod.

## **Bombus hypnorum**

*Bombus hypnorum* has a short proboscis<sup>4</sup> and a rounded head. They usually have a uniformly ginger coloured thorax, a black haired abdomen and a white tail. Some individuals with extended yellow coloration exist. Their body can have a length from 11 millimeters to 18 millimeters [58].

## **Bombus lapidarius**

*Bombus lapidarius* is a red-tailed bumblebee and is distinguished by its black body with red markings around the abdomen. Males also have a yellow band around the abdomen and yellow markings on the face. This species also has a medium-sized proboscis [59].

## **Bombus lucorum**

This species has a short proboscis. The predominant color is black, with a pale yellow collar, a yellow band on the second tergite and a white tail. The individuals of this species have an length from 12 millimeters to 22 millimeters [60].

## **Bombus pascuorum**

The thorax of *Bombus pascuorum* is either yellowish or reddish-brown in color. It has gray hair on the first four abdominal segment and yellowish or reddish-brown hairs on the fifth and sixth terga. The body length may vary between 12 and 32 millimeters depending on the sex of the individuals and depending on its role, namely queen or worker. This species has a long proboscis compared to other bumblebees species [61].

## **Bombus pratorum**

*Bombus pratorum*, also called early bumblebee has a small length that may vary between 10 and 17 millimeters. The individuals of this species have a yellow collar, a yellow first tergite and a red colouration on the tail. The head of *Bombus pratorum* is rounded and the proboscis is short [62].

## **Bombus terrestris**

*Bombus terrestris* individuals are 11 to 22 millimeters long. They look like *Bombus lucorum* but the yellowish bands of *Bombus terrestris* are darker [43].

---

<sup>4</sup>Proboscis: Elongated appendage from the head of an animal.

## Vespula vulgaris

*Vespula vulgaris*, also known as common wasp is the only species of interest that does not belong to *Apidae* family. It belongs to *Vespidae* family. It measures between 12 and 20 millimeters. This species has yellow bands on its prothorax and black dots and rings on its abdomen. This species can also be distinguished from other *Vespula* species by a single black mark on its clypeus<sup>5</sup>.

## A.3 Conclusion

To identify the different arthropod species, the entomologists use a classification tree and observe the different parts of the insects' body. The tree classifies the insects from general to specific categories. This system includes the order, the class, the family and finally the species. Each of these insect's categories present visual specificities in common. The precise identification of *Hymenoptera*'s species can be done by observing the colors or the size of the insects.

---

<sup>5</sup>Clypeus: One of the sclerites that make up the face of an arthropod.



(a) *Apis mellifera*



(b) *Anthophora plumipes*



(c) *Bombus hortorum*



(d) *Bombus hypnorum*



(e) *Bombus lapidarius*



(f) *Bombus lucorum*

Figure A.4: *Hymenoptera* species - part 1 [31].



(g) *Bombus pascuorum*



(h) *Bombus pratorum*



(i) *Bombus terrestris*



(j) *Vespula vulgaris*

Figure A.5: *Hymenoptera* species - part 2 [31].

# Appendix B

## User manual

This appendix describes the project architecture and the implementation details that are necessary to understand how to use our program on a Raspberry Pi. First of all, and as mentioned previously, our entire project is available on GitHub:

- The dataset and the weights: [https://github.com/vahidbe/Master\\_Thesis\\_Bees\\_Dataset\\_and\\_Models](https://github.com/vahidbe/Master_Thesis_Bees_Dataset_and_Models)
- The project's repository: [https://github.com/jothibautucl/Master\\_Thesis\\_Faster\\_RCNN](https://github.com/jothibautucl/Master_Thesis_Faster_RCNN)

### B.1 The Bees Dataset and Models repository

This repository is organized as shown in figure B.1. The *trap\_results* folder contains the images that were taken by our trap. These images were not used in the training nor in the validation of the algorithm. It is divided in two sub-folders: the *sample* folder containing a sample of our data where insects are visible, while the *raw\_results* contains a folder for each model, with the raw results of the detection algorithm (by raw, we mean that there has been no filtering of these images, that there can possibly be a big amount of images that are not relevant, that do not contain insects, or that contain insects of different species than the ones represented by the classes of our model). This encompasses the raw images, the cropped images that were produced from them and the processed images containing the bounding boxes and the labels over each of the detected insects. There is also a CSV file corresponding to each set of detections that summarizes the insects that were encountered. The *training\_data* folder contains all the images that were not produced by us (coming from **gbif** [32] and **observations.be** [a2014\_observationsbe]), and that were used in the training, the validation or

in the testing of our model. Notice that this data is not partitioned in three sets as this is done at run-time and saved in the configuration.

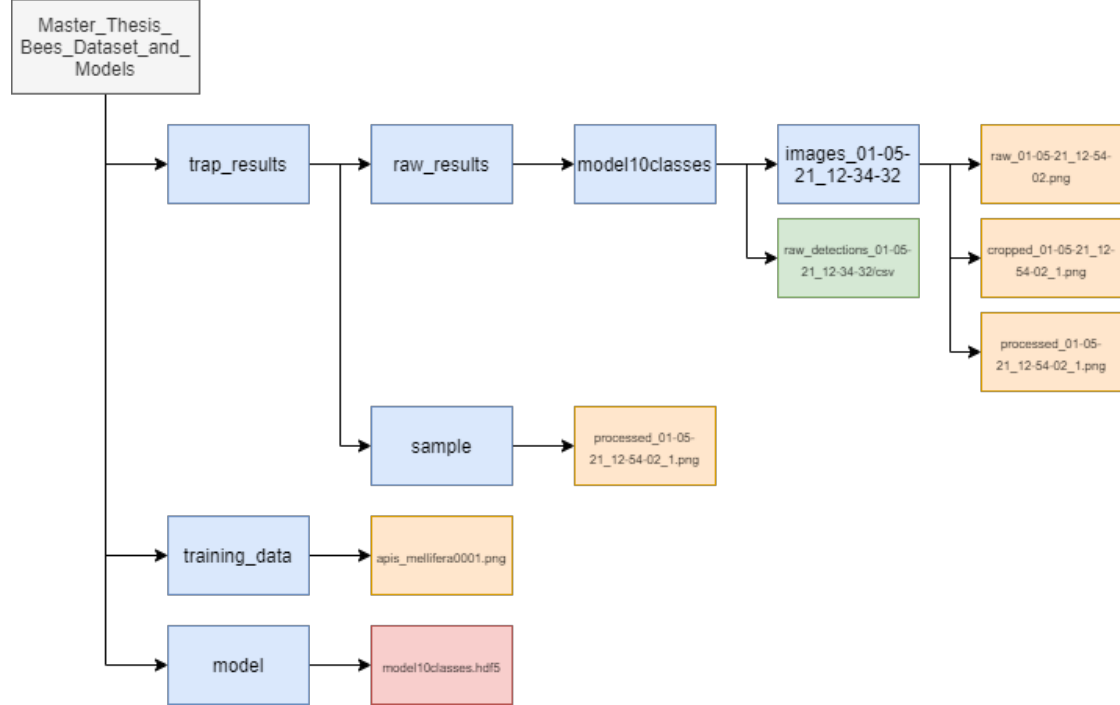


Figure B.1: Structure of the master's thesis Bees Dataset and Model repository.

## B.2 The Faster RCNN repository

This repository contains all the necessary scripts to run the training, the validation and the evaluation of a model, as well as the program to run on the Raspberry Pi for the automated trap.

### B.2.1 Project layout

The organization is represented in figure B.2. These scripts are present directly in the root folder, along with the necessary folders. The *config* folder stores the configurations of the models, as well as the partition of the dataset in the three sets necessary for the training, the validation and the assessment. The weights of our models are saved in the *model* folder. The *other* folder stores a wide panel of information. The visualizations and summaries of the training and validation are stored in the *logs* sub-folder. It contains a sub-folder for each model validated, as

well as a training summary (in a CSV file). The sub-folder contains the weights, summaries and loss representations for each of the different validation phases. The *other* folder also contains a *histogram* sub-folder containing the histograms representing the dataset that can be generated with our *data\_visualization.py* script, a *graphs* sub-folder that contains all graphs generated in the evaluation of the models, and a *output files* sub-folder that contains the output of the training and/or inference scripts. The other sub-folders of the *other* folder are the *powerBI* and *tables* sub-folders that contain our Power BI reports and the data on which they rely respectively. The repository also contains a *predictions* folder containing the inference results generated by the *frnn\_test.py* script, a *results* folder that is the place where the trap saves its detections, with the same structure as in the *trap\_results* folder in the dataset and weights repository, and finally a *resources* folder gathering all the additional resources needed for the full functioning of the scripts. More specifically, a user that wants to use the *add\_meteorological\_data.py* script needs an API key to the **OpenWeatherMap** interface.

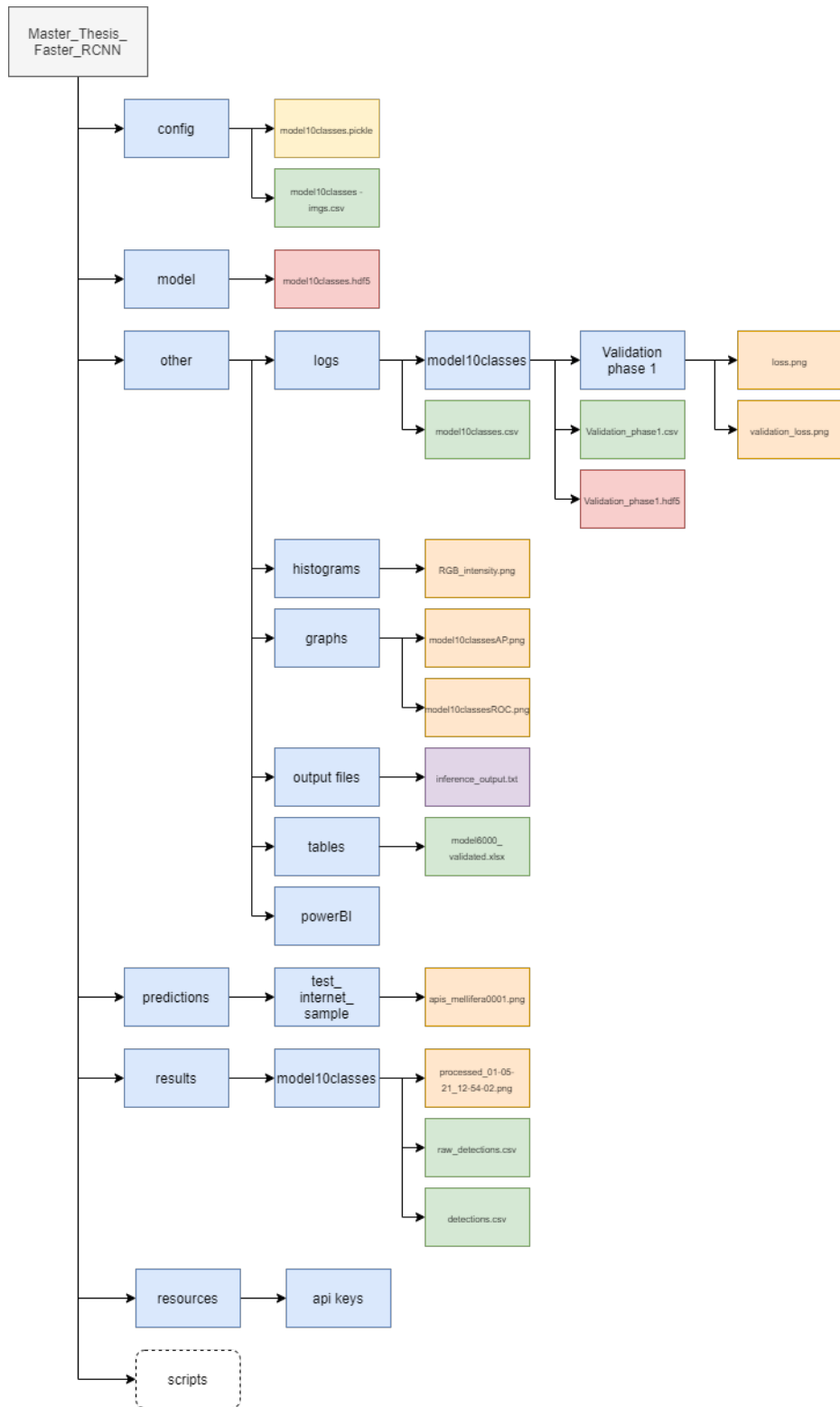


Figure B.2: Structure of the master's thesis Faster RCNN repository.

## B.2.2 Power BI reports

The detections produced by the trap using the *raspberrypi\_real\_time.py* script located in the *results* folder, after being parsed by the *add\_meteorological\_data.py* script can be displayed using our Power BI report in the *powerBI* folder. That report is based on the *detections.xlsx* table of the *tables* folder, meaning that the data from the detections needs to be copied into the corresponding table of that folder. The provided report presents some basic information that can be exploited from this data and gives an overview of what is possible to see using our automated trap. This report is shown in figure B.3.



Figure B.3: Power BI report using our trap data.

## B.2.3 Code organization

The actual scripts in the repository belong to one of two following categories:

- The scripts that allow the training, validation and testing of the algorithm, that help these tasks in any way, or that help analyze the data.
- The scripts that enable running the detection and the processing on a trap, or that help process the data generated by that trap.

## Training, validating and testing scripts

The interactions between the different scripts of this category, the folders described above and the data is represented in figure B.4. There are three main scripts: the *frcnn\_train.py* script that handles the training and validation of new models; the *frcnn\_test.py* script that can assess the performance of a model, or use it to produce detections from new images; and the *data\_visualization.py* script that can analyze the dataset and produce visualizations.

The *frcnn\_train.py* script uses the dataset to produce a partition (the training, validation and testing sets) in the data, which is saved in a CSV file in the *config* folder. It also saves a configuration pickle file in that folder that represents the configuration object linked to the model. The script outputs various information about the optimal loss or the progression of the validation, that can be saved in the *output\_files* folder. The actual summaries and graphs of the loss throughout the training/validation are saved in a recorder object using the *recorder.py* script, which saves the data in CSV files, and the graphs in PNG files in the *logs* folder.

The *frcnn\_test.py* script has two main functionalities: processing external data or evaluating a model. The first uses new images that are not contained in the dataset, and make detections on it by using a provided model. It reads the HDF5 file from the *model* folder in order to load the weight of that model, and the pickle file from the *config* folder in order to retrieve its configuration. The processed images with the detections are saved in the *predictions* folder. The second additionally reads the CSV file from the *config* folder in order to retrieve the data partition, and so the testing set for the evaluation. The script gathers the detections on the images of that set as well as their annotations, and compare them to produce the AP and ROC curves of the model using the *plot\_graphs.py* script, and save them in the *graphs* folder. Both *frcnn\_test.py* and *frcnn\_train.py* rely on the *libraries.py* file for their shared functions.

The *data\_visualization.py* script produces histograms based on frequency of the pixel values, or the sizes or ratios of the images of the dataset and save them in the *histogram* folder as mentioned above.

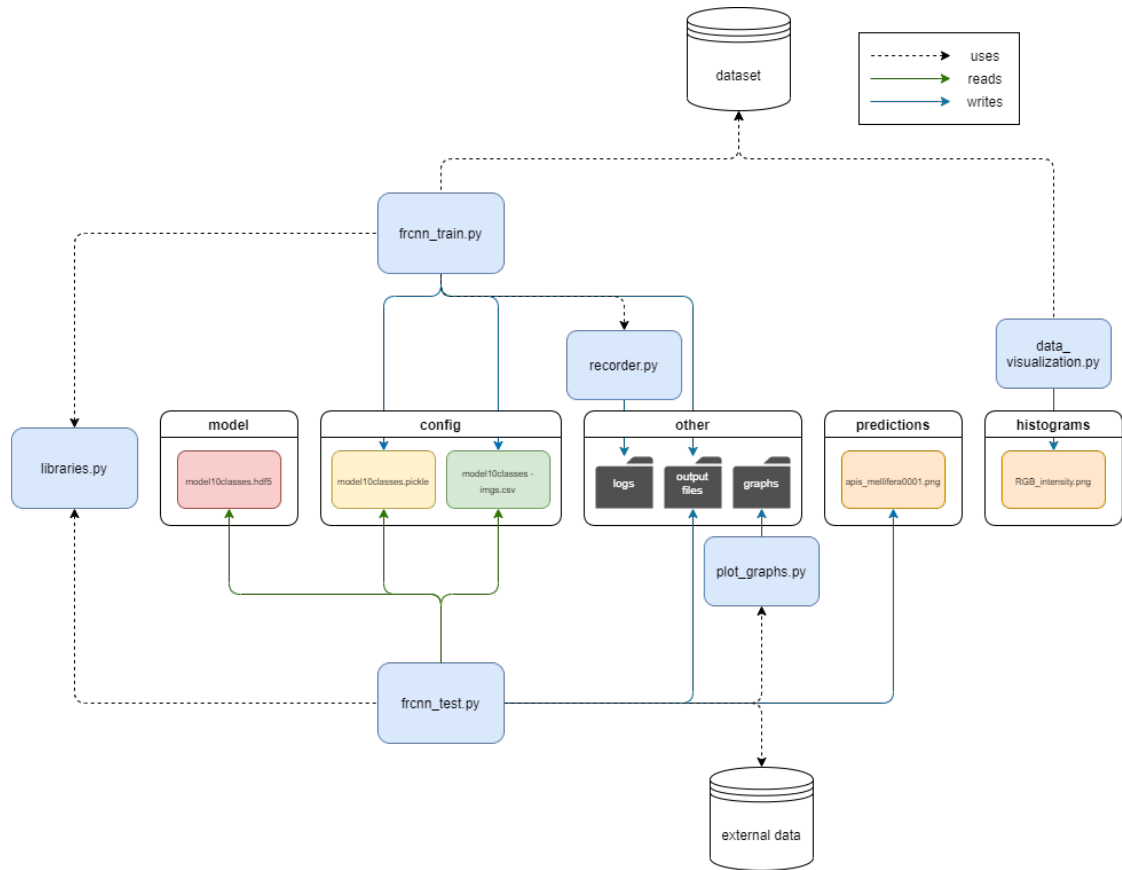


Figure B.4: Training, validating and testing scripts' organization.

### Trap related scripts

The scripts related to the real-time detection and processing on the trap are interacting as shown in figure B.5. The main scripts are *raspberrypi\_real\_time.py* that handles the detection of insects as well as their classification, on the Raspberry Pi; and *add\_meteorological\_data.py* that parses the results of the previous script and adds weather data corresponding to the time the detections were taken.

The first uses a given model, which means that it needs to access its weights in the HDF5 file stored in the *model* folder, as well as its configuration pickle file in the *config* folder. Note that it does not need the data partition as it runs the processing on data that the trap takes using a camera commanded by the *real\_time\_libraries.py* script. This script contains the functions corresponding to the two processes of the *raspberrypi\_real\_time.py* script: the detection process that detects movement and commands the camera to take pictures, and the processing process that uses the *detection\_libraries.py* file to produce the model's detections

and their classification. The detection process can also use the *stepper\_motor.py* file to command a direct current (DC) motor (to open or close a trapdoor on the physical trap). It saves the raw images in the *results* folder. The processed images are saved there too by the processing process, and are also logged in the *raw\_detections.csv* file.

The second uses the *weather\_collector.py* file to request the meteorological data to the **OpenWeatherMap** application programming interface (API), for each of the detections saved in the *raw\_detections.csv* file in the *results* folder. The new detections file with the weather data is saved in the same folder, in a CSV file called *detections.csv*. This operation requires an internet connection.

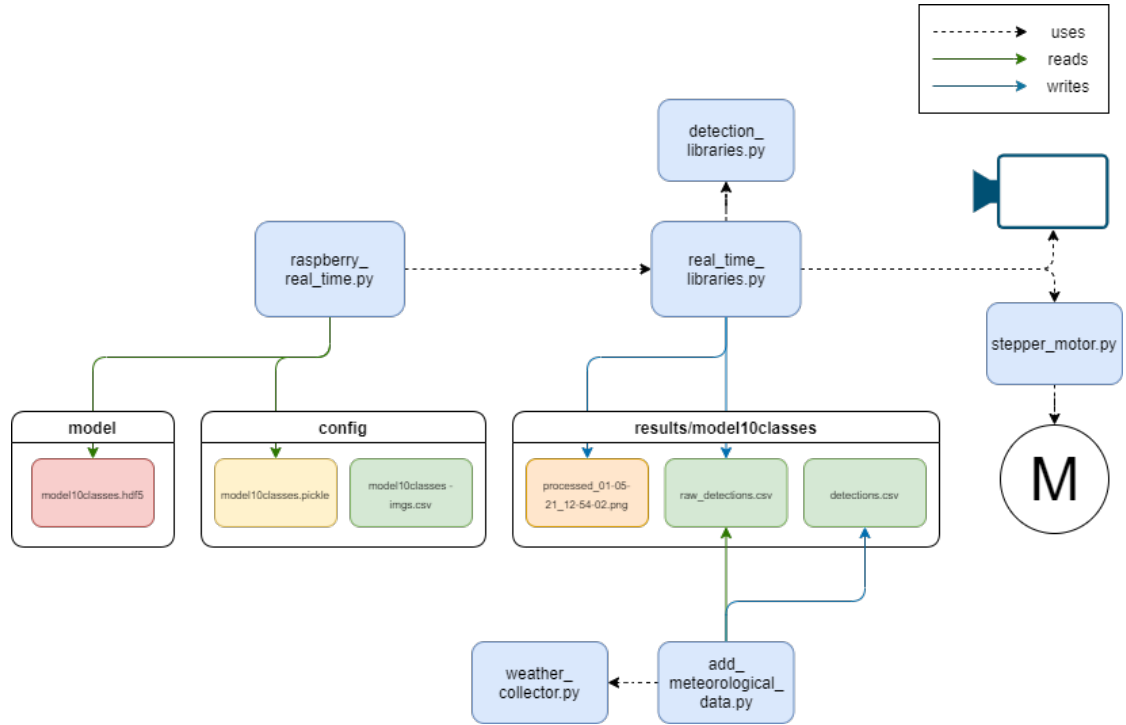


Figure B.5: Trap related scripts' organization.

## B.2.4 Running scripts

The scripts described previously can be run using Python as described below.

### **frcnn\_train.py**

As mentioned previously, this script allows the user to run the training or the validation of a new model. The different parameters of this script are the following:

- **model\_name**: this corresponds to the name of the model that is trained. Each model must have a unique name. This name is used to produce or access the configuration, weights, results or other files related to this model.
- **dataset**: the path to the full dataset.
- **annotations**: the path to the annotation (.txt) file corresponding to the dataset.
- **num\_epochs**: the number of epochs, of times that the training set is entirely traversed during the training or during a validation phase.
- **weights**: the initial weights on which the model restarts its training (VGG16 pre-trained weight on the Open Images Dataset by default).
- **validation**: this is False by default. If the user wishes to validate a model instead of simply training it, it must be set to True.
- **validation\_code**: if the previous parameter is set to True, a validation code is given for each validation phase. If the validation is to be resumed from (excluding) a previous validation phase, the code of that phase is to be given in this parameter.
- **use\_gpu**: set to True if the user wishes to use the GPU for the training/-validation of the model, False otherwise.

A simple training can be triggered as such:

```
python3 frcnn_train.py
--model_name="model10classes" \
--dataset="../data" \
--annotation="../data/annotation.txt" \
--num_epochs=8 \
--validation=False \
--use_gpu=True
```

A model validation requires modifying the script to enter the list of hyper-parameters that need to be validated, as well as their values' field. The validation can be started as follows:

```
python3 frcnn_train.py \
--model_name="model10classes_validated" \
--dataset="../data" \
--annotation="../data/annotation.txt" \
--num_epochs=8 \
```

```
--validation=True \
--use_gpu=True
```

### **frcnn\_test.py**

The testing script can be used in two manners: to assess the quality of a model, or to produce detections based on external data. Its parameters are listed here:

- **model\_name**: this corresponds to the name of the model that is used. Each model must have a unique name. This name is used to produce or access the configuration, weights, results or other files related to this model.
- **dataset**: the path to the full dataset, or the data folder on which the evaluation or the detections are to be done.
- **annotations**: the path to the annotation (.txt) file corresponding to the dataset (can be left to None if the user does not need to evaluate the quality of the model).
- **from\_csv**: if set to True, the evaluation or the detections is done on the images in the CSV file of the *config* folder corresponding to the testing set. Otherwise, all the images in the dataset that are annotated is used.
- **process\_images**: if set to True, the script makes detections on the images provided and save them in the *predictions* folder.
- **processed\_directory**: this corresponds to the name of the directory that is created in the *predictions* folder if the previous parameter is set to True.
- **evaluate**: if set to True, the script evaluates the quality of the model.
- **trap\_images**: if set to True, the script evaluates the quality of the model based on the trap images.
- **noise\_reduction**: this parameter allows the user to use a noise reduction preprocessing technique. The different choices implemented are 'box\_filter' and 'gaussian' and are described in section 5.3.2.
- **histogram\_equalization**: this parameter allows the user to use the histogram equalization preprocessing technique. It is described in section 5.3.2.
- **gamma\_correction**: this parameter allows the user to use the gamma correction preprocessing technique. It is described in section 5.3.2.

- **use\_gpu**: set to True if the user wishes to use the GPU for the training/-validation of the model, False otherwise.

An inference on some images can be triggered as such:

```
python3 frcnn_test.py
--model_name="model10classes" \
--dataset="../trap_results" \
--from_csv=False \
--evaluate=False \
--process_images=True \
--processed_directory="trap_results_processed" \
--use_gpu=True
```

The evaluation of the performance of a model can be started as follows:

```
python3 frcnn_test.py \
--model_name="model10classes" \
--dataset="../data" \
--annotation="../data/annotation.txt" \
--from_csv=True \
--evaluate=True \
--process_images=False \
--use_gpu=True
```

### **data\_visualization.py**

This script aims at visualizing both training and testing data to compare them. Its different parameters are :

- **split\_img\_file**: this corresponds to the path to the CSV file that contains split train/test/validation dataset.
- **ratio\_area**: If set to true, the script plots the histogram of the ratio of the box area on the picture area.
- **ratio\_width\_height**: If set to true, the script plots the histogram of the ratio of the box width on the box height.
- **rgb\_intensity**: If set to true, the script plots the histogram of the red-green-blue (RGB) intensity of pictures.
- **gray\_intensity**: If set to true, the script plots the histogram of the gray intensity of pictures.

To observe all histograms on the dataset that contains all ten classes, the script can be run with the following command :

```
python3 data_visualization.py \
--split_img_file='./config/model10classes - imgs.csv' \
--ratio_area=True \
--ratio_width_height=True \
--rgb_intensity=True \
--gray_intensity=True
```

### **raspberrypi\_real\_time.py**

This script is designed to be called on the Raspberry Pi. Its different parameters are:

- **model\_name**: this corresponds to the name of the model that is used. Each model must have a unique name. This name is used to produce or access the configuration, weights, results or other files related to this model.
- **bbox\_threshold**: the threshold of probability over which a detection is considered and displayed.
- **demo**: if set to True, the script uses the webcam of the user to run detections in real-time. It can be run on a computer and not only on the Raspberry Pi (False by default).
- **show\_images**: if set to True, the camera feed and the stream of processed images are displayed on the screen.
- **resolution**: the resolution of the pictures taken by the camera, in the following format: "(width, height)".
- **fps**: the number of images taken per second by the camera (only images where movement was detected are saved).
- **use\_motor**: if set to True, the detection process communicates with the motor to close or open a trapdoor on the physical trap to respond to movement in front of the camera.
- **use\_gpu**: set to True if the user wishes to use the GPU for the training/-validation of the model, False otherwise.

The real-time insect detection can be triggered as such:

```
python3 raspberry_real_time.py \
--model_name="model10classes" \
--bbox_threshold=0.6 \
--show_images=False \
--resolution="(1920, 1080)" \
--fps=2 \
--use_motor=False \
--use_gpu=False
```

The demo can also be started as follows:

```
python3 raspberry_real_time.py \
--model_name="model10classes" \
--bbox_threshold=0.6 \
--show_images=True \
--demo \
--fps=2 \
--use_gpu=True
```

### **add\_\_meteorological\_\_data.py**

The *add\_\_meteorological\_\_data.py* script adds weather data to the provided detections CSV file and takes the following parameters:

- **model\_name**: this corresponds to the name of the model, and it is used to recover the path to the raw\_detections of the model. Each model must have a unique name.
- **lat**: the threshold of probability over which a detection is considered and displayed.
- **lon**: if set to True, the script uses the webcam of the user to run detections in real-time. It can be run on a computer and not only on the Raspberry Pi (False by default).
- **input\_file**: the filename of the CSV file that contains the raw\_detections to which the user wants to add weather data. This file must be present in the sub-folder corresponding to the model\_name, in the *results* folder.

The user can add meteorological data to a set of detections by running the following command:

```
python3 add_meteorological_data.py \
--model_name="model10classes" \
--lon=50.5251 \
--lat=4.6107 \
--input_file="raw_17-03-2021_15-01-18"
```

### **bbox\_validation.py**

The *bbox\_validation.py* script allows the user to verify the bounding boxes of the annotation file in order to remove the incorrect ones. Each bounding box is displayed and the user is able to choose between pressing **Enter** to validate it or **Escape** to discard it. The arguments of this script are listed here:

- **dataset:** the path to the full dataset.
- **input:** the path to the initial annotation (.txt) file corresponding to the dataset.
- **output:** the path to the corrected annotation (.txt) file, the output of the script.
- **startfrom:** the line number in the input annotation file where the user desires to start the checking.

The user can verify the bounding boxes in his annotations by running the following command:

```
python3 bbox_validation.py \
--dataset="other/trap_results" \
--input="other/trap_results/annotations.txt" \
--output="other/trap_results/valid_data_annotations.txt" \
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN  
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | [www.uclouvain.be/epl](http://www.uclouvain.be/epl)