UCLouvain

École polytechnique de Louvain

Automated nocturnal insect monitoring in real-time using YOLOv4

Authors: **Gilles Charlier, Simon Hick**
Supervisors: **Olivier Bonaventure, Renaud Detry**
Reader: **Maxime Istasse**
Academic year 2020–2021
Master [120] in Computer Science

Contents

Glossary

backbone The network used in a CNN to extract features. 14, 19, 26, 30, 31, 33, 52, 54, 55, 57, 58

bounding box Object localisation represented as an outline of a rectangle in a image. In numerical format coordinates and measurements of the box are used. 28, 48, 49

classifier A classifier predicts the class of some data, e.g. recognising the species of a single insect in a picture. 4, 10, 18, 19

ground-truth bounding box Bounding boxes labelled by hand in the training/ validation and testing set. Indicates the exact location of an object in an image. 15, 28, 48, 49, 50

heatmap Technique used in object detection to represent the spatial distribution of annotations in a dataset. 28

Intersection-over-Union An evaluation metric used in object detection algorithms calculated by dividing the area of overlap between a bounding box and its ground-truth box by the area of their union. 15, 48, 52

lepidoptera Insect order that includes moths and butterflies. 38, 39

mAP From the acronym mean average precision. This metric is used in object detection models to evaluate their performance. Explained in detail in chapter 5. 11, 16, 51, 54, 55, 60, 63

object detection Computer vision technique that consists of locating and classifying object instances within an image or a video. 2, 3, 15, 17, 18, 20, 21, 31, 34, 42, 43, 48, 51, 52, 54, 63, 64

segmentation Image processing technique aiming to partition an image into sets of pixels under certain constraints (i.e. foreground and background). 4

Acronyms

BoF Bag of Freebies. 19, 20

BoS Bag of Specials. 19

CNN Convolutional Neural Network. 3, 4, 5, 8, 9, 10, 11

CSL Classification Score Layer. 14

FC Fully Connected. 8

FCN Fully Convolutional Network. 13, 17

FPS Frames Per Second. 20, 43, 63

GBIF Global Biodiversity Information Facility. 23, 24, 25, 27, 37, 47, 63, 77, 78

JN Jetson Nano. 43

RoI Region of Interest. 13, 15, 17

RPi Raspberry Pi. 43

RPN Region Proposal Network. 4, 13, 14, 15, 17, 18

SSD Single Shot Mutlibox Detector. 52

TLT Transfer Learning Toolkit. 2, 22, 30, 31, 32, 34, 70

Abstract

The sharp decline in insect population during the last decades has highlighted the need for more insect monitoring. Until recently, acquiring data on insect population was done manually and required a significant amount of human workload. To lessen this load, new methods of data collection have been developed using computer vision. Automated systems can recognise insects with the help of AI and monitor the fluctuation of biodiversity in a location. These systems usually consist of a trap, or a drone, to take pictures of the insects. Those pictures are then sent to a central server that performs the recognition.

In our thesis, we chose a different approach by performing every aspect of monitoring directly in our trap, i.e. attracting, taking pictures of and recognising insects. With this difference, we had the possibility to make a completely autonomous trap, as it will work offline. We propose a real-time autonomous monitoring trap that uses YOLOv4 as its object detection model with a ResNet18 backbone. The model has been trained and tested on more than 48000 images taken from GBIF. Our experimental results show that detection on the edge is possible in real-time and with an mAP of 85.33% with ResNet18.

Acknowledgements

We would like to express our thanks to all the people without whom this thesis would not have been possible. First and foremost we are extremely grateful to our supervisors Prof. Bonaventure and Prof. Detry.

Prof. Bonaventure guided us throughout this thesis by helping us whenever we had questions about the research or writing process, by steering us in the right direction and by providing us materials for our first prototype. Prof. Detry gave us invaluable theoretical and practical advice about existing object detection algorithms as well as constant feedbacks on our approach. He shared with us his knowledge and passion for the artificial intelligence without hesitation.

We would also like to thank all the people who collaborated in some way on our thesis; Claire Stulemeijer for her precious advice on entomology and her methodology for studying insects, Günter Hick for his thorough proofreading, Alexandre Fiset for 3D printing parts of our first prototype, P. Spileers for sharing his passion for nature observation with us. And last but not least, our parents and close friends who always had a profound belief in our work and our abilities.

Chapter 1

Introduction

In recent years, changes in the insect population has been much debated. The declining number of pollinators, for example, has been widely advertised thanks to multiple awareness campaigns led both by governments and NGOs alike. Unfortunately, this change in population size is a worrisome trend that is not limited to pollinators and can be observed in the world's total insect population size. It does not mean that all insect species are going extinct, as for example some species like freshwater insects may be multiplying [25], but the general trend seems to point towards a steep decline [14].

Although this population decline is widely accepted, its severity is still heavily debated; Sánchez-Bayo and Wyckhuys (2019)[54] suggest that 40% of all insect species may disappear in the next decade and predict catastrophic consequences in the absence of serious measures. But in direct response to this publication, Simmons et al. (2019)[56] notes that, although the decline is an important issue that merits more public awareness, Sánchez-Bayo and Wyckhuys's review has several issues creating a bias favouring alarmist results.

All these publications underline the importance of monitoring the world insect population. Without accurate and worldwide data, it is hard to draw any precise conclusion. Additionally, currently there is only a limited number of species being monitored on a large scale, such as butterflies, dragonflies, bees and moths [13]. Consequently there is a need for more extensive systems, capable of recognising more members of the Insecta class.

Given this context, the goal of our thesis is thus to provide an answer to the current need for insect monitoring. Our solution focuses on nocturnal insects as

they can be attracted by light, discarding the need for pheromones to get them to enter our trap. Our system should be able to identify multiple orders of insects through a camera, track them as they enter its field of vision, count each individual only once and keep in a log any identification made.

A network of autonomous traps would be able to give information about the evolution of insect population in multiple geographical points, including remote ones. Knowing where the sharpest declines are occurring, would hopefully enable us to better isolate the source of the problem since the cause of the general decline is not yet known [13].

The first step required to build this trap is to develop a recognition system capable of identifying the captured insects. There are multiple models available to perform this kind of task. We present how they work and which models interest us in the state of the art in chapter 2. To train the model of our choice, we decided to use the Nvidia Transfer Learning Toolkit. It allows us to test the accuracy of different models easily, so that we may select the one best suited for our needs. The training process is explained in more detail in chapter 3. We obtained the best results with YOLOv4 [1], which we accordingly have used for our final configuration.

Once a good recognition model has been elaborated, it can be deployed on the trap. Our trap is comprised of a camera to capture pictures of the insects, an embedded device to run the object detection model and multiple LEDs to attract insects at night. A more complete presentation of the trap is given in chapter 4 and a complete list of the components is given in Appendix B.

Our YOLOv4 model achieves a mean average precision (mAP) of 85.33% using images of insects taken online. Regretfully, the unusually cold weather we had this spring resulted in a very low number of nocturnal insects up to the month of June. This tempered greatly both with the collection of in-the-field pictures and our ability to properly test the effectiveness of our trap. However we still had the opportunity to make multiple measurements of our model's accuracy, which are shown in chapter 5.

Finally and as is usual, we end this thesis with a conclusion presenting what we achieved and the future works that could be built upon it.

Chapter 2

State of the art

The task of insect classification is not a new one, the protection of agricultural lands from pests has been a vital effort for millennia. This can still be seen today, as most of the publications about insect recognition are still about pest detection rather than classifying insects to monitor biodiversity. For this reason, most of the papers we present in this state of the art are oriented toward pest detection. Nevertheless, they are still relevant to establish the methods available for insect recognition.

Nowadays we are more reliant on automated insect recognition than ever before, due to the declining number of entomologists [17] and the rise in popularity of IoT systems. And thus, multiple methods of automated insect recognition have been developed in the last decades.

Initially, automated insect classification was done with hand-crafted features. A hand-crafted feature is a characteristic known to be useful to recognise a certain insect by entomologists that is included in the recognition model. But recently, the field of computer vision seems to be oriented toward the use of a Convolutional Neural Network (CNN) [24]. A CNN is able to learn the features by itself but requires more data to be trained. The recent popularity of CNN is due to multiple factors, the most important one being its performance.

However, classification is not enough to monitor insects. Since multiple different insects might be present on a single image, we have to find the location of each insect and classify them separately. This is called object detection.

We present in this chapter two families of object detectors. The first, called region-based, are two-stage detectors. Two-stage detectors start by finding the location of every object in an image and then classify each object found. In opposition, the second family called one-stage detectors perform the localisation and classification in a single step. In this family, we present the model called YOLO, short for *You Only Look Once*.

2.1 Insect classification: from traditional methods to CNNs

We can represent a generic pipeline to create an insect recognition system through the following steps [73] (represented in Figure 2.1):

1. **Prepare the database:**

   Classification models require a significant amount of images to be properly trained. These images should be labelled so that each one is assigned to a class. When using handcrafted feature extraction (ie. not using a CNN), it is more common to expect a lab-based setting with insects pinned in a standard pose because of the difficulty of recognising insects in different angles [33].

2. **Image processing:**

   Images in the database must be processed to be usable by the classifier. Removing artefacts off of our pictures and normalising luminosity, for example, are very common pre-processing techniques to make images easier to use [21]. Usually, it is also a good idea to use segmentation to find which zone of the image is important and which is not, such as the background. However, when using a CNN it is not obligatory to perform segmentation on our images beforehand, as different ways to locate objects have been developed using only CNNs, for example RPNs which we present in detail in subsection 2.3.3.

3. **Feature extraction and selection :**

   To classify our pictures, the classifier uses features to recognise which insect is present. These features represent different aspects of an insect, for example its shape, its colour or its texture. For this step, it is important to use any taxonomy expertise available to know what characteristics can differentiate each species, so that we might handcraft specific features that are less generic than the colour of the insect for example [21]. There are many possible

features and since some of those can be heavily demanding to compute, it is important to choose the most relevant ones. This is known as feature selection.

It is in feature extraction and selection that we can find another big difference with the CNN. A CNN is able to learn by itself to identify features and also what features are important when recognising an insect. Thus we can say that expert knowledge in taxonomy is less important than before, given that the CNN is able to find features without any human intervention.

4. **Classification :**

   There are two main differences in the classification; first concerning the performance, where the CNN seems to deliver better accuracy than its counterpart and secondly, in the required amount of images to get the best results, as the CNN needs a significantly larger amount of training images to work at its best.

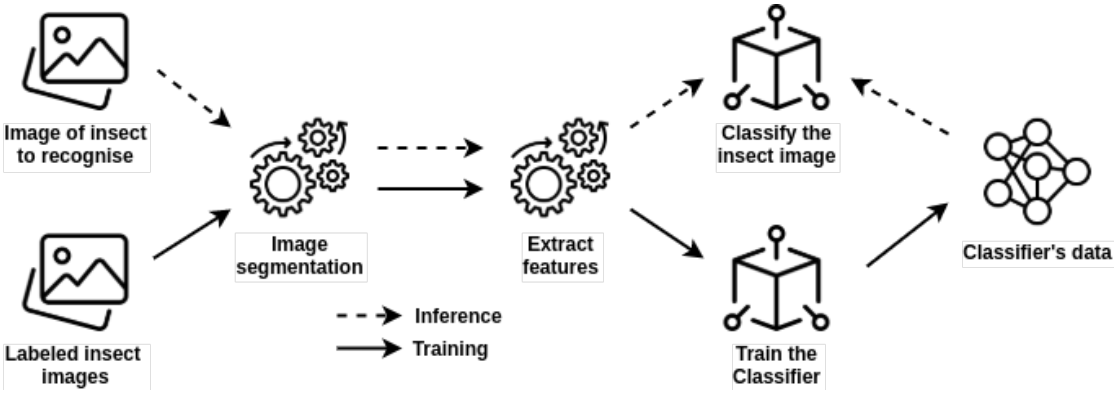


Figure 2.1: Generic pipeline used for non-CNN recognition models

2.2 Introduction to Convolutional Neural Networks

Introduced by Yann Lecun in 1998 [29], the Convolutional Neural Network is a type of neural network often used today in image recognition and analysis tasks. It is specialised in data processing with a grid-like topology. It is made of five different types of layers.

2.2.1 Convolution layer

This first layer receives an image as input. It analyses the input image thanks to filters (or kernels) which extract precise features. For example, filters can be sensitive to sharp, curvy edges or textures. As a result, the network can learn which filters detect features. They are applied with a sliding window through the matrix of pixels and produce as output a 2D matrix for each filter applied, in the case of a grayscale image [44]. If using coloured images, we would end up with three 2D matrices, one for each colour in the red-green-blue scheme (RGB). We continue our explanations using grayscale images as the process is the same as with colours, except the fact that every step has to be repeated thrice to be applied to each of the three matrices.

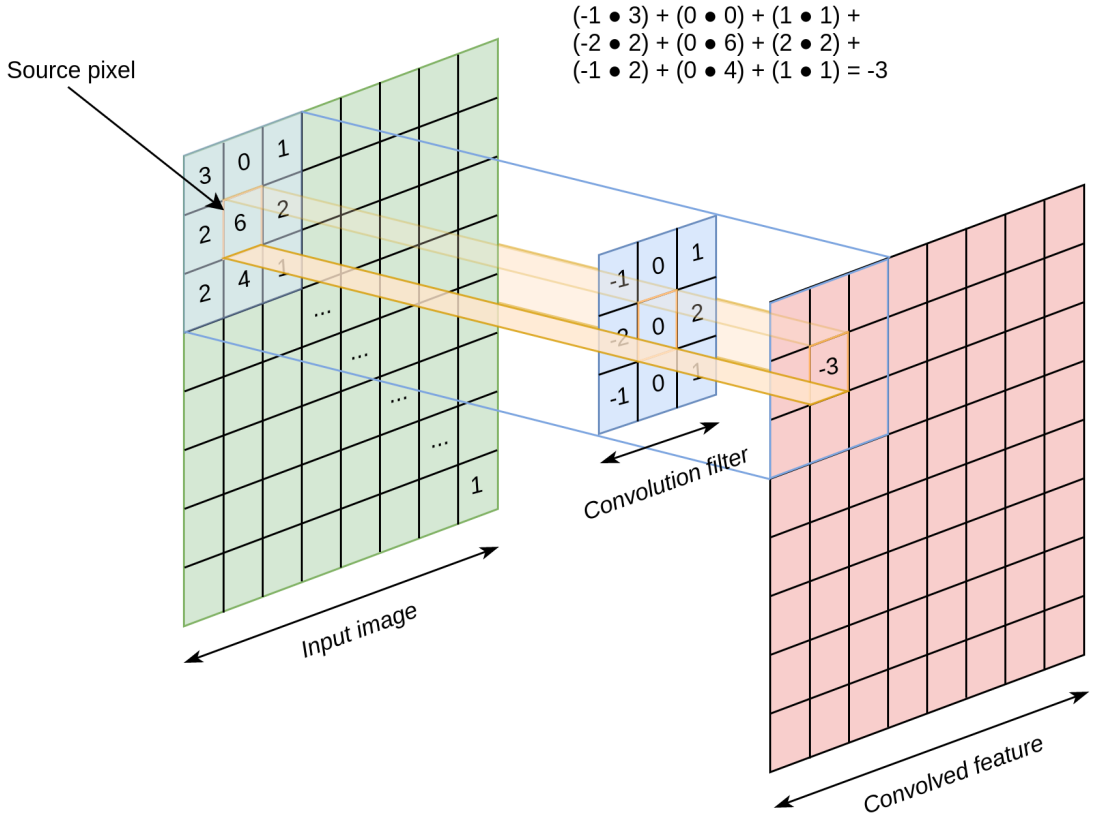


Figure 2.2: Convolution process

2.2.2 Activation function

The activation function is a node that is inserted in the middle or at the end of the network to decide if a neuron should be activated or not. This function is really useful to map the input signal to an output signal. The most commonly used function is the `ReLU` function (Figure 2.3) which maps any negative input to zero [52] and gives the ability to the network to learn non-linear functions, which are really important in image analysis [63].

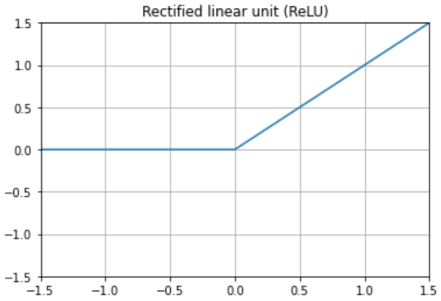


Figure 2.3: ReLU function

If we have multiple classes, there seem to be two choices for our activation function: either softmax or sigmoid. If the classes are mutually exclusive, the best choice is softmax since it gives the probability distribution, which means all output sum to 1.

Once the activation function has been applied, all we have to do is to select the class with the highest probability and we are done. While for the sigmoid, we could choose to select all classes with an output value greater than a certain threshold [71]. If we want to make the distinction between multiple insects orders and choose the most appropriate one, softmax seems to be the better choice.

2.2.3 Pooling layer

When using feature maps produced by the convolution layer, we see each detail of each feature from each convolutional layer, but we are not able to detect higher-level patterns. We would need to stack together a great number of output results of the convolutional layers to learn these patterns. This is called spatial hierarchy [5]. But spatial hierarchy is very expensive and that is why pooling has been created. Pooling is an operation used for down-sampling features by reducing the spatial size of the network [63].

The pooling layer reduces the number of parameters and computations in the network, and therefore reduces the chances of over-fitting. Two operations are used at this step; average pooling and maximum pooling. Max pooling takes the largest element from the rectified feature map while average pooling takes the average.

2.2.4 Flattening layer

To prepare the data for the next step, we have to flatten the 2D matrix we get after max pooling. This means we put the values of the matrix into a single vector. This is needed for the data to be usable by the fully connected layer.

2.2.5 Fully Connected layer

After having flattened the feature maps, we can feed the FC layer which is the final layer of the CNN. The last FC layer produces the output of the whole network.

2.2.6 CNN strengths and weaknesses

An important difference with non-convolutional neural networks is the sparse interaction property. In CNNs, a neuron of a layer does not have to be connected to every neuron of the next and previous layers. Since a pixel of an object only needs to be strongly connected with other pixels of that same object, there is no need to have strong connections between layers. This reduces the number of parameters to fine-tune and therefore the complexity of the network.

The use of convolutional layers followed by max pooling layers also brings an important advantage called translation invariance. Due to the way the dimensions of the feature map are reduced after max pooling, some positional loss is going to occur. This means that the same object present on two images but shifted by a few pixels has no impact on the feature map that is outputted. Compared to non-CNN networks which can struggle with this difference, translation invariance brings more flexibility to the CNNs. This effect of max pooling is illustrated on Figure 2.4

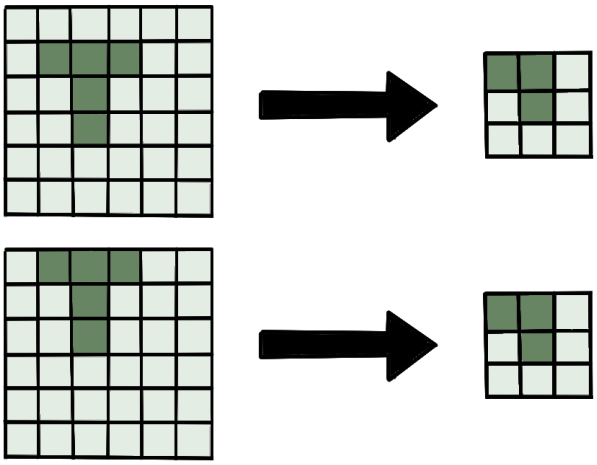


Figure 2.4: The result of a 2x2 Maxpool: a small translation has no impact on the feature map

We should note that translation invariance has its limits. If a model is trained to detect an object that is only ever present on the upper left corner of a wide image, it will have a negative impact on recognition if the object is on the opposite corner at testing [20]. In this case, a better dataset should be made and/or data augmentation should be used. For a description of data augmentation, see section 3.4.

The use of max pooling solves another similar problem that arises from the convolutional layers. When low-level features are combined into higher-level features by the convolutional layers, no information about positional relationships between features is passed. For this reason, no difference would be made between a disassembled chair and an assembled one (see Figure 2.5). Max pooling helps to solve this problem by reducing the spatial size of the data.

2.3 CNN architectures used in insect recognition and detection

There is a multitude of architectures for CNN, but only a few ones have been used to recognise insects. In the next paragraph, we explain some of the most important architectures in chronological order of publication, by presenting their additions to the fields of classification and detection. And when possible, we will talk about publications that use those architectures for insect recognition.

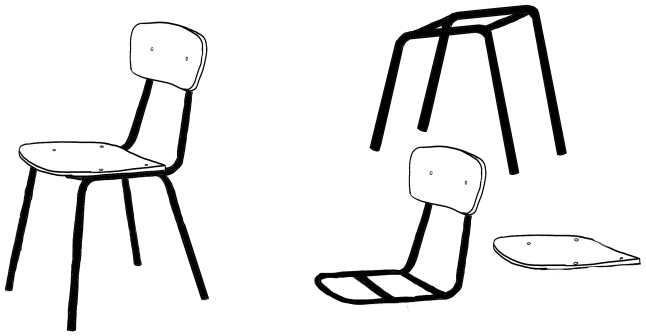


Figure 2.5: A CNN may classify those two images as chairs without the use of max pooling.

2.3.1 Classifiers and CNNs

Since our system must detect and recognise more than one insect at the same time on an image, a classifier is not suitable for this task. Indeed, a classifier can identify which insect is on an image, but cannot determine where this insect is and if there is more than one. We have found some articles that use CNN as their major components such as proposed by Rustia et al. [53] that describes how to implement a multi-class insect pest identification method or Lim, Kim, and Kim [30] that explains the performance effect of a CNN insect classifier. In these articles, the images contain only one insect at a time. If segmentation is needed to split images that contain multiple insects, it is made during the preprocessing step of the database [53]. The sliding window detector is another technique used in CNNs. This approach is effective in terms of speed [8, 11], but requires that all objects have the same scale, which is not the case for insects. Therefore, we are not using a simple CNN architecture, but an improvement of it which we will present later on.

2.3.2 Illustration of transfer learning on VGG

The VGG model is a good example of a simple CNN. Developed by the Visual Geometry Group [57], its two most famous variants are the VGG-16 and VGG-19

thus called after their number of layers; 16 and 19 respectively. A representation of the VGG-16 architecture can be seen on Figure 2.6. VGG-16 achieved a record accuracy score in the ILSVRC-2014 competition [28], after being trained for weeks on high-end GPUs [15].

This is its main flaw, it is very slow to train, requires substantial computational power and huge training data sets. To face this problem, M. Valan et al. [65] used a VGG16 network pre-trained on ImageNet dataset, and re-trained it using a bit less than 100 images per class. Thanks to this method called transfer learning, they have been able to reach good results with a small amount of images per class. This is due to the fact that early layers of CNNs learn simple features like edge detection, so these layers can be used to recognise completely different objects. The trained weights of a CNN are used as a basis for a new one, and then during its training, this new network is tuned to detect new classes.

According to M. Martineau, training only the last layer of the VGG-16 to recognise insects is enough to achieve good results [34]. It was also able to tackle the unbalanced learning problem even if only marginally. This problem arises when the class distributions are highly imbalanced, ie. when some insects are more present in the data than others.

It is important to note that these last two publications are made with images of insects taken in lab settings. If their results are very informative regarding the methods used, we can not expect to reach the same conclusion with our in-the-field insect recognition model. For example, 100 images per class are not enough in our case. However transfer learning and class balance are two important notions that we made sure to keep in mind during this work.

2.3.3 Region-based

Introduced in 2014 by Rosh Girshick and his team [12, 11], region-based CNN (R-CNN) is presented as a combination of region proposals and CNN (see Figure 2.7). The principle is as follows; the system takes an input image, generates around 2000 region proposals using selective search, computes features for each proposal using a large convolutional neural network and then classifies each region using class-specific linear SVMs. R-CNN improves mean average precision (mAP) by more than 30% over the best result on VOC 2012 at that time.

Here is a quick explanation of how selective search generates region proposals: It performs an initial segmentation dividing the image into small regions. Then

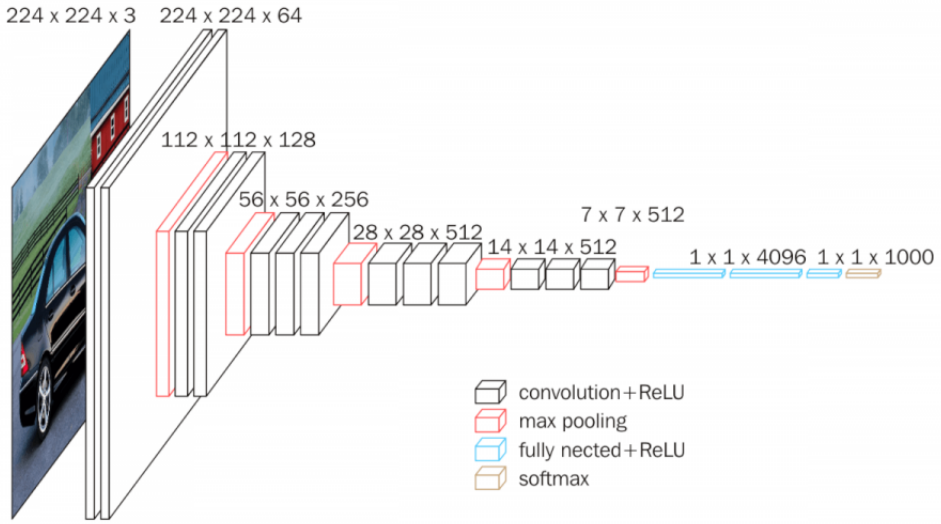


Figure 2.6: Representation of the VGG-16 architecture [66]

it will recursively combine similar small regions into bigger ones. The similarity between regions depends on multiple factors: colour similarity, texture similarity, size similarity and fill similarity. The regions found with this method are then used to generate candidate object locations. We also note that since it is used as an external module independent of the detectors, any other method that generates region proposals can be chosen.

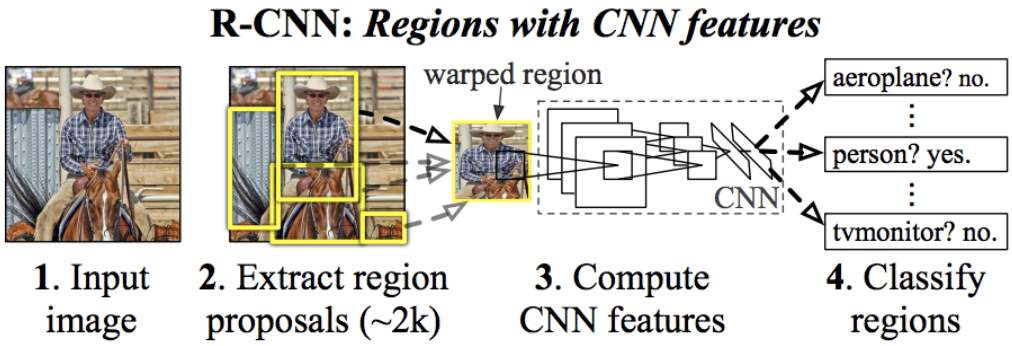


Figure 2.7: Overview of R-CNN execution [12]

2.3.4 Fast R-CNN

In 2015, Girshick proposes a new algorithm called fast R-CNN aiming to solve some of the drawbacks of R-CNN [10]. Mainly, as its name suggests, the advantage of the new algorithm is its speed. Instead of extracting 2000 region proposals, the convolution operation is done once per image and a feature map is generated from it (see Figure 2.8). From this feature map, multiple region proposals are identified. Those regions are then changed into squares with a fixed size so that they can be sent to a fully connected layer. Fast R-CNN outputs the expected class and the bounding box of the Region of Interest (RoI).

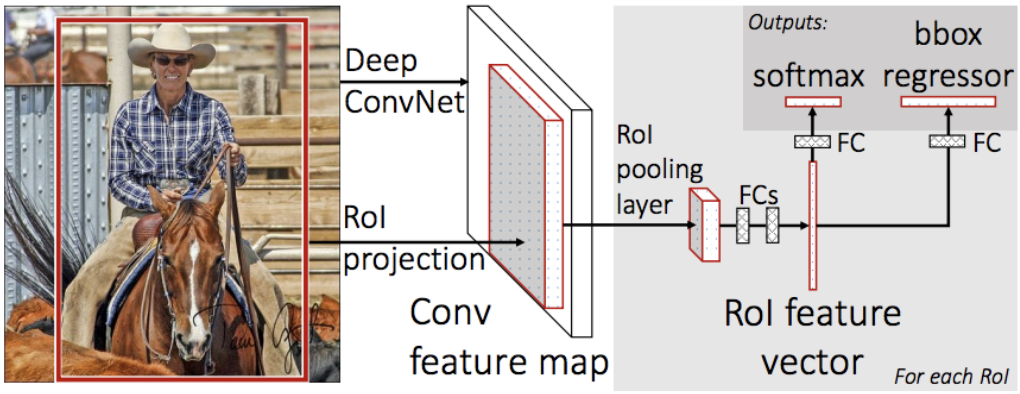


Figure 2.8: Fast R-CNN organisation [10]

2.3.5 Faster R-CNN

Faster R-CNN expands on Fast R-CNN by replacing the selective search used to get region proposals since it is a time-consuming process. Faster R-CNN is composed of two modules, the first is an Region Proposal Network (RPN) that finds regions in the image where interesting objects are and the second uses the proposed regions to classify the image within the proposed regions in the same way as Fast R-CNN. Being based on Fast R-CNN, Faster R-CNN has also two outputs for each candidate object, a class label and a bounding-box offset [51].

Region Proposal Network (RPN)
RPN is an Fully Convolutional Network (FCN) that predicts at the same time the object coordinates as well as their objectness score. This network takes feature maps produced by the convolutional layers as input and learns which regions are

more interesting in the image by performing a binary classification. Faster R-CNN fixes the bottleneck of Fast R-CNN by using an RPN instead of using the selective search to find regions. This improvement speeds up the training process by around 10 times according to the original paper [51].

First, the RPN takes a feature map produced by the last convolutional layer of the backbone as input. To produce region proposals, the RPN slides a rectangle window on a feature map as shown in Figure 2.9.

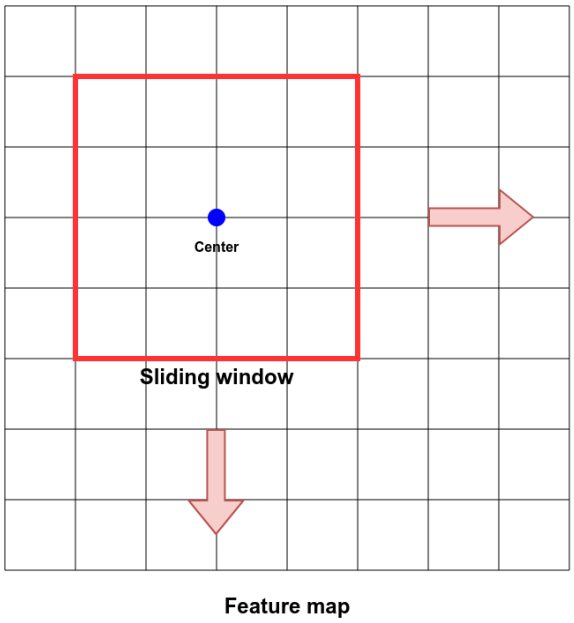


Figure 2.9: Sliding window over feature map

For each window position, multiple region proposals (also called anchors) are generated following some parameters like anchor size or ratio. Anchors are always generated in the centre of the sliding window as we can see on Figure 2.10.

Once anchors have been generated, the RPN trains itself to classify properly anchors that contain foreground or background as well as finding their offset. Offsets are used to translate foreground anchors.

If the RPN receives a feature map of size `M x N` pixels, it outputs two layers named Classification Score Layer (CSL) of size `(M x N x 2k)` where `k` is the number of bounding boxes produced per position (in Figure 2.10, `k=9`) and is multiplied by 2 because it is a binary classification between foreground and background. The other layer produced is called reg layer (regression coefficients for boxes layer) is of size `(M,N,4k)` and corresponds to the position offset of each anchor.

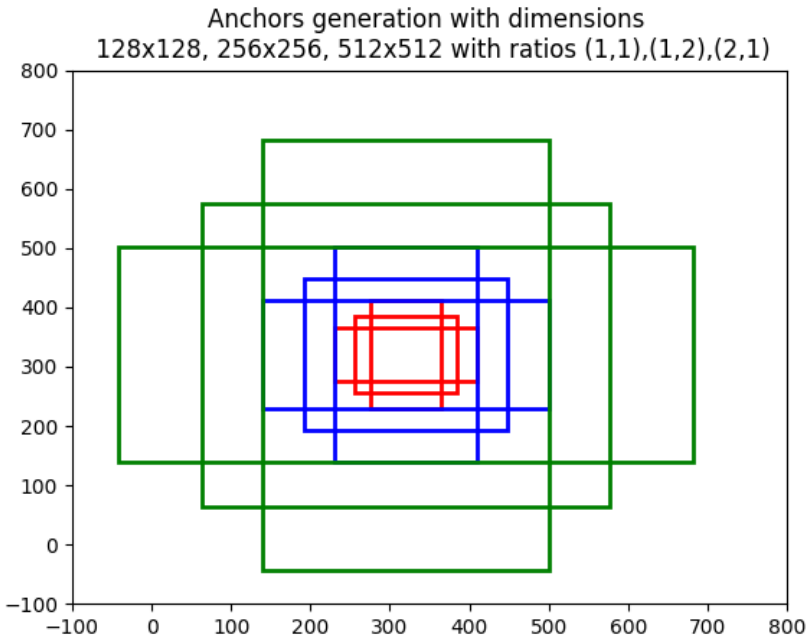


Figure 2.10: Anchors generation centred on (320,320)

The RPN needs to be trained, which means that it has a loss function based on binary class labels that have been assigned to anchors that have a high Intersection-over-Union with any ground-truth bounding box. Once the candidate boxes have been outputted, the non-maximum suppression (NMS) filter redundant boxes [31].

Now that the RPN has finished its job, it is time to classify the predictions. Faster R-CNN extracts feature maps of Region of Interests using pooling as explained in 2.2.3, and classifies each RoI based on their features. For classification, Faster R-CNN uses the flattening principle as well as fully connected layers. We already explained those concepts in subsection 2.2.4 and subsection 2.2.5. At the end of this process, Region of Interests with their class are output by the algorithm.

Balakrishnan Ramalingam achieved good results in the field of insect detection by using a pre-trained ResNet50. The ResNet50 is much deeper than the previously presented VGG16 which is a clear advantage in insect recognition since the desired detection is much more complex than for regular object detection process [47], but it is longer to train and slower during inference.

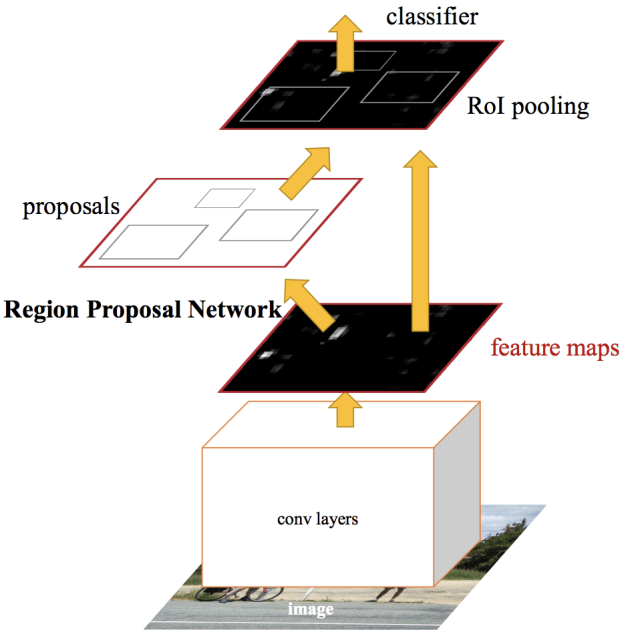


Figure 2.11: Faster R-CNN and the RPN [51]

Nieuwenhuizen, Hemming, and Suh [39] used Faster R-CNN to detect tomato whitefly and its predatory bugs on yellow sticky traps. Combined with a digital single-lens reflex camera or a smartphone camera and controlled light conditions, they achieved a weighted averaged accuracy of 87.4% with a correlation of hand-counted insects and deep learning counted insects of over 0.95. To attain this level of performance, they used 1350 image slices and a model pretrained on the MS-COCO dataset retrained during 200.000 epochs.

More recently, Karar et al. [23] used Faster R-CNN combined with cloud computing to accomplish the recognition task of insect pests. They proposed a model achieving a mean average precision of 98% over five species on macro photos from various sources. These species are very different from each other and the photos are of very good quality, which explains these results. This model is meant to be used on a smartphone application and therefore requires an internet connection.

Xia et al. [72] used another approach by designing an improved version of a convolutional neural network based on VGG19. With this method, they achieved an mAP of 89.22% on field conditions with 0.083 sec per inference image and a training time of 11 hours, which is 7 times lower than Fast R-CNN. Unfortunately, they did not compare their model with newer models like Faster R-CNN that drastically

reduces the training and inference time. Authors used on average 70 training images from Google per class for 24 species, which is relatively low compared to other approaches.

2.3.6 Mask R-CNN

Mask R-CNN is a combination of Faster R-CNN and an FCN and performs at the same time object detection and segmentation (see Figure 2.12). This framework was proposed by Kaiming He et al. in 2017 as an extension of Faster R-CNN, and therefore contains almost the same two stages. The first stage is an Region Proposal Network (RPN) which proposes candidate object bounding boxes. These boxes are then feeding the second stage which consists of finding their class and their offset. In parallel, Mask R-CNN also takes the `RoIAlign` layer output and predicts a binary mask for each Region of Interest (RoI) [35] with an FCN. `RoIAlign` is a variation of `RoIPool` used in Faster R-CNN that fixes the misalignment issue between the extracted features and RoI by using an unquantized RoI stride [16]. As a result, authors claim to obtain pixel-wise masks for objects in the image, as well as their bounding boxes and class name [16]. One huge downside is that Mask R-CNN needs pixel-wise masks as annotations for input images which is way more time-consuming to create. As far as we know, Mask R-CNN is not used yet for insect recognition, publications generally use faster R-CNN instead.

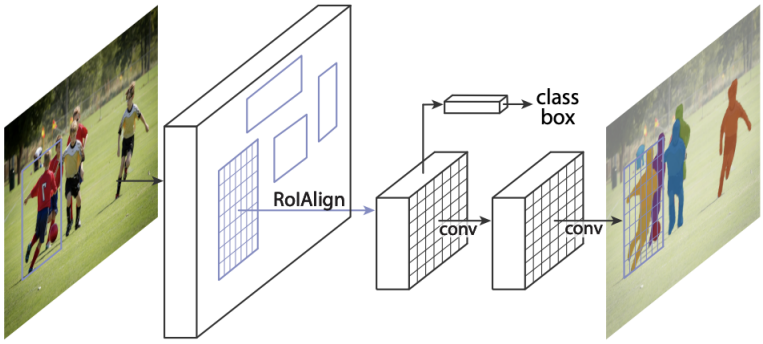


Figure 2.12: Mask R-CNN architecture [16]

2.3.7 One stage detectors

The R-CNN series of object detection models we have presented above are part of the family of two-stage detectors. As we saw, these two stages are (1) an RPN proposing regions of interest and (2) a classifier processing the region candidates.

In opposition, the one-step detectors forego the region proposal step and perform detection in a single stage.

YOLO: You Only Look Once

A well-known one step detector is called YOLO. Which is short for *You Only Look Once*, YOLO got its name because it will only pass over the image once to do its prediction instead of, as in the case of Faster R-CNN, multiple times for the various regions in an image. Thanks to this improvement, YOLO is capable to do inference in real-time.

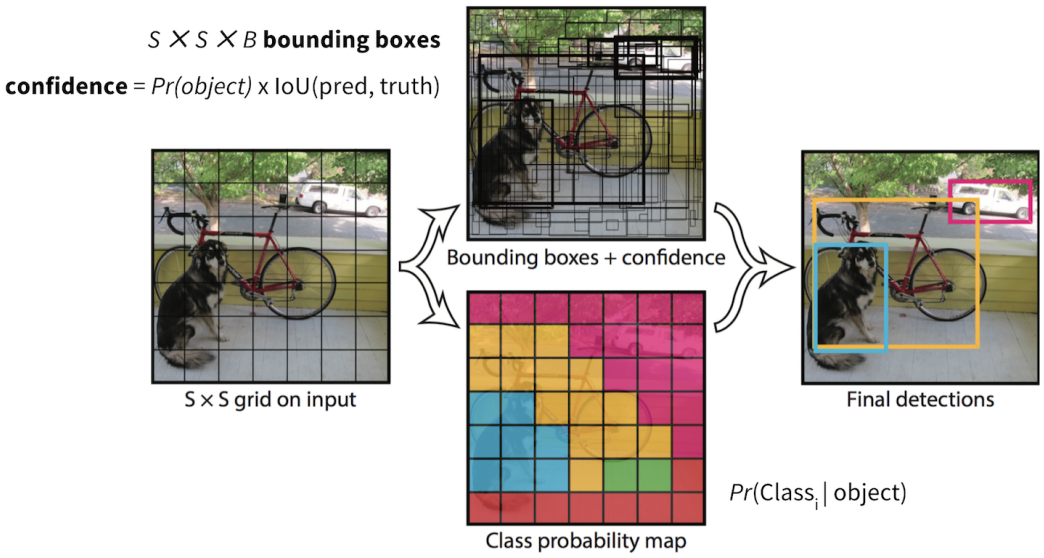


Figure 2.13: The model's workflow [50]

It works as follows:

- The model is based on a pre-trained backbone, like the ones used in region-based detectors. For example, VGG16. But the final layer of this backbone is modified to output a tensor of the size required by YOLO, which is given and explained below.
- The image to be processed is split into $S \times S$ cells (see Figure 2.13). Each cell is independently capable to predict if it contains the centre of an object or not. They will also predict a confidence score indicating the likelihood that the cell contains an object. If the cell contains an object, it predicts the probability of this object belonging to each class. If a cell contains the centre of the object, it will also predict the coordinates of the bounding box of said object. These predictions are encoded in a tensor of size $S \times S \times (B * 5 + C)$. B is the number of bounding boxes per cell, 5 represents the four box coordinates and the confidence score, and C represents the number of classes to be recognised by the model.

From YOLO to YOLOv4

YOLO immediately attracted the attention of the computer vision community when it was published in 2015. It has since been incrementally improved upon multiple times and still is being improved today. We will not give here a list of all the improvements made in the last years, but each version has brought many small changes.

For example, YOLOv2 introduces the use of anchor boxes like in faster R-CNN, instead of the bounding boxes used previously. The fully connected layers previously used to predict the bounding boxes are replaced by convolutional layers. These anchor boxes sizes will be determined by running k-means clustering on the training data priors to the training to determine good dimensions [48].

Another improvement, made for YOLOv3, is that it drops the use of the softmax activation function, to use an independent logistic classifier for each class instead. This is used so that labels are not mutually exclusive anymore [49].

YOLOv4 introduces the concepts of Bag of Freebies (BoF) and Bag of Specials (BoS). BoF corresponds to the improvements done before or during training, that will have no impact on inference time. In contrast, to the BoS which takes more computation time at inference, but delivers better accuracy in return [1].

A new method of data augmentation has been used for YOLOv4 and is listed in the BoF. This method is called mosaic and, as its name suggests, consists of combining multiple images into one mosaic. This method will be further explained in section 3.4.

The general workflow of YOLO has not changed since its first version, but a lot of improvements have been made by applying the most innovative ideas of computer vision. Thanks to this, YOLOv3 outperforms Faster R-CNN in a similar environment both in speed and accuracy [58]. And YOLOv4 achieves a similar speed to YOLOv3 but with a big improvement in accuracy.

YOLO, due to its speed, is an ideal model to do inference at the edge, but it has also other advantages over region-based detectors. Unlike with the use of sliding windows and region-based proposals, YOLO uses the entire image during training time, so it implicitly learns contextual information about classes. YOLO also learns general representations of objects, making it more effective when confronted with new domains, e.g. performing object detection on artwork.

Most of the publications about pest detection that we found are using two step detectors. But after specifically searching for it, we did found a few publications that use YOLO. Wang et al. [67] presented a modified version of YOLOv4 used to detect pests on a sticky trap. The trap is large compared to the size of insects, for this reason the detection model must be capable of recognising very small objects in a large image. To this end, they proposed to use a larger feature map by using a 2-fold upsampling on the feature map at the 8-fold downsampling layer.

Chen et al. [4] compare multiple object detection models to detect pests. When comparing Faster R-CNN, SSD and YOLOv4, it found that YOLO delivered the best accuracy; identifying mealybugs with an F1 score of 100%, Coccidae with that of 89%, and Diaspididae with that of 97%. The detection is made through a mobile app to capture the video while the inference is done in real-time on a server.

Lastly, Takimoto et al. [60] presented YOLOv4 as a region proposal network and EfficientNet as a re-identification method, effectively making a two-stage CNN. As we can expect, this slows down the inference process which no longer is in real-time at around 5 FPS. But the advantage is that YOLOv4 alone has a low precision, which goes from 55% to 89% when using the two-stage version. This publication, unlike the previous ones, uses images taken in the field which are of varying quality. This is closer to the images that are captured by our trap and is thus giving us a possibility of a solution if we have a precision too low with our YOLOv4.

2.4 Conclusion

In this chapter, we have seen how insect detection has been practised over the last 20 years. The first automated approaches to insect recognition used hand-crafted features or CNNs that could only classify images of a single insect. To be able to recognise more than one insect in an image, several methods to divide the image into small regions were developed. On the one hand, one-stage detectors considered object detection as a regression problem such as YOLO, while two-stage detectors split the detection and classification tasks. In general, the latter are more accurate but slower than the one-stage detectors.

Even if choosing the most recent approach looks like the best option, there are some downsides. For example, Mask R-CNN is the latest region-based model, but instance segmentation requires a lot of extra work to annotate images with masks. Since our system's goal is to recognise insects and report their presence at a precise geographic point, a mask is not needed. Its previous version called Faster R-CNN performs object detection as well as Mask R-CNN without outputting masks and is, therefore, a better choice. But two-stage detectors are slow and less accurate compared to one-stage detectors like YOLOv4. Since we aim to do real-time inference with the best accuracy, we have chosen to use YOLOv4.

From existing publications, we have seen that all approaches take place in daylight, some use a sticky trap to ensure that insects stay in place for sharper photos, some use smartphone cameras to perform the recognition task on the go. Our approach is therefore interesting since none of them uses an "insect-friendly" trap that spare insects after their identification and all of them require an internet connection or a human action if they are deployed in the field.

Chapter 3

Insect recognition

3.1 Global view

This chapter describes how we made our insect recognition pipeline from start to finish. Figure 3.1 shows an overview of how the pipeline is organised. We start from the data collection, which is comprised of building a dataset of insect images and the annotation of said images, i.e. writing down the coordinates and the taxonomic order of every insect present in all images. Then, we present how we trained our YOLOv4 model with the Transfer Learning Toolkit (TLT) and the advantages/disadvantages it induces. We end with the framework that runs the inference in our trap and generates the logs in real-time: DeepStream SDK.

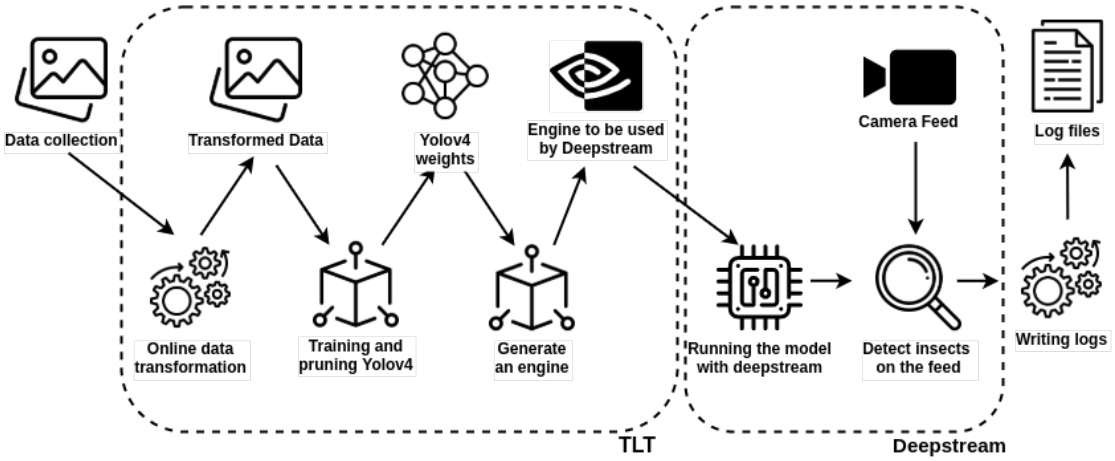


Figure 3.1: Overview of the pipeline

3.2 Data collection

Data format and quantity have always been two of the biggest issues when it comes to training deep learning models. Such models and especially convolutional neural networks require a lot of images to become accurate. Fortunately for us, in the case of insect recognition, we can count on images taken by insect enthusiasts around the world, be it professional or captured by amateurs. Since our model is meant to be deployed in and around Belgium, we have chosen to focus on the most common orders and families of nocturnal insects in Belgium and the Netherlands, which are presented in details in section 4.2 *Nocturnal insects*.

As most pictures available online are well made and well-framed, they do not correspond exactly to what we obtain with our trap. Still, training with those good quality pictures paired with a perfect focus on the insect is useful to learn precise features. The downside is that it is not representative of the pictures we capture with our trap. To remedy this problem, we made a modification to the data we collected online. We used multiple methods of data transformation to lessen the differences between the pictures found online and those made by our trap; more details about this in section 3.4 *Data augmentation*. A second modification that was planned was to add images taken by our trap into the training set, but because of the bad weather as explained in chapter 6, this improvement has not been made.

3.2.1 Global Biodiversity Information Facility (GBIF) and other online databases

After some research looking for a good database, we ended up discovering multiple websites, each hosting millions of insect images. The most complete we found is called Global Biodiversity Information Facility (GBIF). In their own words, *GBIF —the Global Biodiversity Information Facility—is an international network and data infrastructure funded by the world's governments and aims to provide to anyone, anywhere, open access to data about all types of life on Earth.* In fact, GBIF gathers also images from most of the other websites we found, such as *observation.be* and *waarnemingen.nl*, which are the largest collaboration platforms for flora and fauna observations in Belgium and the Netherlands respectively.

GBIF contains more than 1.5 billion occurrences of observations of all kind of life forms around the globe. These occurrences do not all correspond to images taken, but may just be a report of a sighting of an animal or a plant. In the Benelux

alone, around 2.1 million pictures of insects are available for our model to train on.

This database can be used via an API, or via the website itself. We have chosen to generate the list of URLs of each order/family manually via the website since the website classifies insects species with a tree representation, which allows us to explore the insect's taxonomic tree and learn what insect is present in large quantity and which is not in the Benelux. To generate a list of images URLs, we had to select some filters. Here are the common filters we chose for each insect classes:

- **Basis of record :** This field represents the kind of observation performed if it is a human observation, a machine observation with a photo trap or a preserved specimen. Since our system must recognise living insects in their natural environment, we have chosen only human observation.
- **Countries :** We selected Belgium and Netherlands as our reference countries. We have decided to add the Netherlands because it has a huge community of observers and its fauna, flora and climate are not that different from Belgium.
- **Media type:** Insect recognition can be also done via analysis of high-frequency sounds but for this thesis, we are only using images.
- **Occurrence status:** We only want images that contain insects so this filter is set to present.
- **Taxon key :** This filter represents the scientific name of the species, order or family. This filter varies according to the desired insect class.
- **Life stage:** Since caterpillars and larvae are not attracted by light and are hard to recognise, we decided to only classify insects based on their adult form also called *imago*.

Once the file containing all the URLs is obtained from GBIF, we parse it to download the images we want. More information about our custom parser and its performances are available in our section A.3. The links to the files containing the insect's occurrences are also available in section C.1.

3.2.2 Data collected in the field

Even if our model is trained with around 5000 images per class, it is always good practice to add images captured in the field in real conditions. These photos are

often worse in terms of quality but are similar to those that we captured with the trap. These pictures are useful to train the model to be responsive even with bad pictures like we would get during in-the-field settings. We have chosen to use embedded devices such as a Raspberry Pi or a Jetson Nano and we implemented a motion detection script inspired by two sources. The first source is from Arducam's CSI IMX477 camera documentation [46] and the other from a blog that explains how to implement a motion detection system with a Raspberry Pi [59].

In a video stream, and especially at night due to the low luminosity, there are always small variations between frames. To prevent itself from reacting to these little variations, the script first resizes the image to reduce CPU usage and then applies a Gaussian blur to smooth pixels with their average intensity. Then, after having corrected the focus and light balance adjustments, it begins the motion detection. When something enters the field of view of the camera, it takes a photo only if a significant amount of pixels has changed. This threshold can be set as input of the script.

We also added some other improvements to this script, the most important one being the motion detection algorithm by itself. Where before it compared the current frame with the background frame, now it compares the current frame with the previous one. This adds sensitivity and precision because the script now saves frames as fast as possible during the movement.

Another added feature is the ability to preview the camera in low-resolution mode while taking pictures in full resolution. This allows us to monitor the feed while reducing CPU usage and therefore save energy without compromising the image quality. Of course, the preview is only enabled when a screen is connected.

3.3 Data annotation

Nowadays many data preparation services exist. They offer intuitive interfaces to perform dataset annotation, image preprocessing and augmentation as easily as possible, cutting drastically the time needed to prepare the dataset. But these services are often not free. Thankfully, multiple data augmentation methods are already available in the transfer learning toolkit as explained in subsection 3.4, while data collection is already managed by our parser of GBIF files as explained on subsection 3.2.1. So, the only tool we lack is one to annotate our dataset.

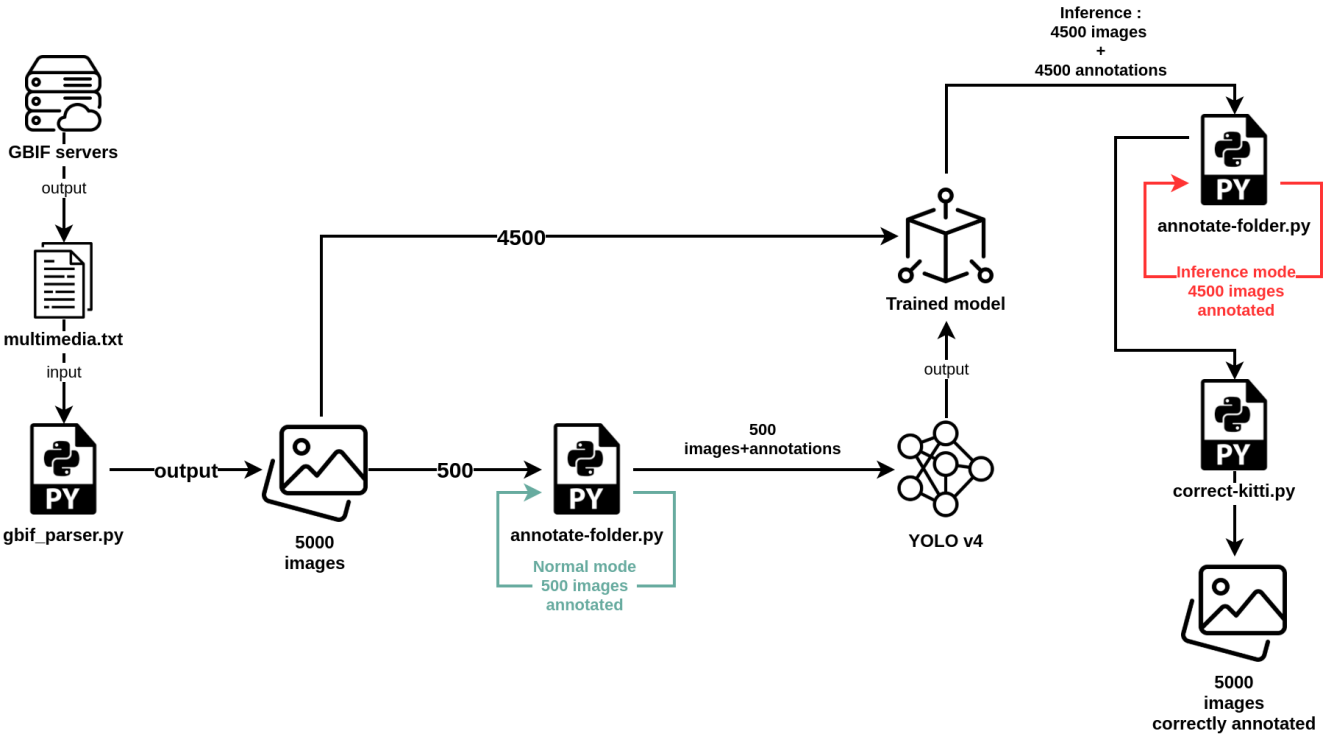


Figure 3.2: Annotation workflow for each class

Given the thousands of images that have to be annotated, finding a good tool is important. We tried multiple tools to annotate images efficiently but no tool was suitable for us, so we decided to fork the GitHub repository `Annotate-to-KITTI` [26]. It is a very basic annotation tool and is thus fast of use, but it still offers multiple shortcut keys making the whole task of annotation simple. For more information about the features that were already implemented in the original repository as well as the changes we made to optimise our workflow, see section A.1.

For faster annotation, we use the workflow shown on Figure 3.2. We first annotate with our forked annotation tool 500 images and then inject them into YOLOv4 with a pre-trained ResNet18 backbone. Then we use this model to annotate 4500 additional images with results of the model inference, which we review and correct when the inference annotations are wrong. With only 500 images YOLOv4 can already perform single class inference pretty well, cutting drastically the time we take for annotation. Table 3.1 shows the number of annotations per class. We repeated this process for each class. These classes and the insects they represent are explained in the subsection 4.2.1 on the next chapter.

Class	Train and validation annotations	Test annotations
Orthoptera	4887	486
Coleoptera	5491	472
Geometridae	4996	485
Trichoptera	4921	438
Noctuidae	5051	484
Diptera	5142	439
Hemiptera	5393	485
Hymenoptera	5035	492
Odonata	5110	482

Table 3.1: Annotations per class

3.4 Data augmentation

When training a model, the training set must be similar to what is observed during inference on the field. But in our case, most of our training set is composed of pictures from GBIF. Generally, those are pictures of good quality with focused on the insect. In contrast, our trap takes pictures of insects that can both appear anywhere in its field of view and be too close to be on focus. So the training set has to be modified to reflect those situations.

Data augmentation can be used to enlarge a dataset. Through simple modifications to an image, we can obtain an altered image that can be added to the training set. For example, by flipping every image vertically we can double the size of our dataset. It is important to keep in mind that these modifications still have to make sense, they should be representative of what our model has to recognise on-the-field. For example, if the goal is to recognise car models, flipping an image vertically is not a good idea since the car ends up upside down, which is not something we encounter often in real-world settings.

In our case, we chose not to use data augmentation to enlarge our dataset but only to modify our images so that they get closer to what is observed in the field. The reason is that we have a large enough amount of images made available to us thanks to GBIF and since insects are very varied, we thought it would be better to annotate even more pictures rather than modifying those we have. For this reason, we have around 5000 pictures per class instead of the 1000 or 2000 we usually see recommended. Of course, these 5000 pictures could also be augmented to have an even larger dataset, but then the training would take an unreasonable amount of time considering the time available to us.

Knowing this, here are the transformations we made to our dataset:

- **Horizontal flip**: each image has a 50% chance of being flipped horizontally.
- **Vertical flip**: the same 50% chance is applied for a vertical flip.

These two transformations are made because we observed that in most of the pictures available, the insects have their heads oriented upward, which is a choice made by the photographers. With these transformations, the orientation of the insects is randomised. Since rotation invariance (i.e. the rotation of a feature does not impact the recognition process) is not build in YOLOv4 [68], this randomisation of direction ensures our model is rotationally invariant after training.

- **Mosaic:** each image has a 50% probability of being used in a mosaic.

To analyse our dataset, we generated a heatmap representing the location of the ground-truth bounding boxes in our images, see Figure 3.3. The bigger the amount of bounding boxes is present at a particular position in every image, the brighter the corresponding position of our heatmap gets. We can see in the figure that most of the bounding boxes are positioned around the centre with a gap of around 100 pixels to the border of the image. As we saw in subsection 2.2.3, translational invariance in CNNs is guaranteed only for small distances. For this reason, we should make sure that insects in our training set are present at every positions. One way to change the uniformity of distribution in our dataset is to use the augmentation called mosaic. It creates a mosaic of 4 random sized images (see Figure 3.4), so that the positions of the objects are spread over more locations. It has also the added benefit of cramming four images into the size of a single one, which forces the model to learn to recognise smaller objects. This is particularly useful in our case since insects are further away from the camera in pictures taken by our trap compared to pictures taken online.

To apply these transformations, we used the online data augmentation provided by TLT. Here *online* means that the changes are applied to the images just before they are fed to the training and thus that our dataset is not permanently modified. To choose which modification we want to apply, we need to enter our parameters into the training's specifications file (more about this file in section 3.5 *Training on a workstation with transfer learning toolkit (TLT)*). Multiple augmentations have a certain probability to be applied to an image. For these, a seed can be set so that the online augmentation stays consistent between two training.

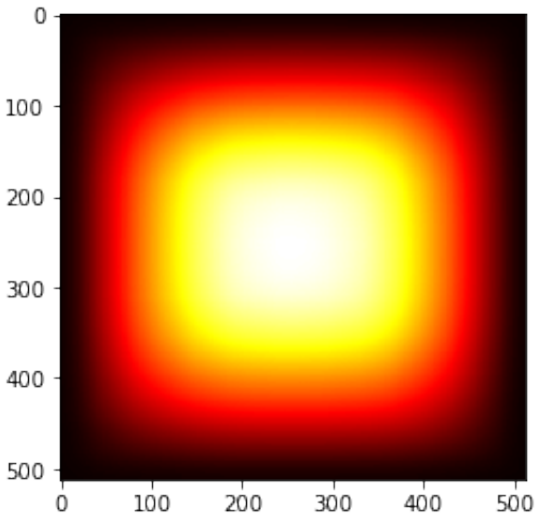


Figure 3.3: Heatmap of the bounding boxes location in our dataset's images

Figure 3.4: Example of a mosaic [27]

3.5 Training on a workstation with transfer learning toolkit (TLT)

NVIDIA's Transfer Learning Toolkit (TLT) is targeted primarily towards businesses, the promise is to speed up development time by simplifying the task of building a custom model.

As its name suggests, it relies on transfer learning to achieve this. TLT provides multiple models and classification backbones pre-trained on the Open Image Dataset to use as a base for our own model. Since TLT also provides scripts to perform data augmentation and pre-processing, what is left for the user to do is: acquiring and analysing a database, and then tuning the augmentation, training and inference parameters.

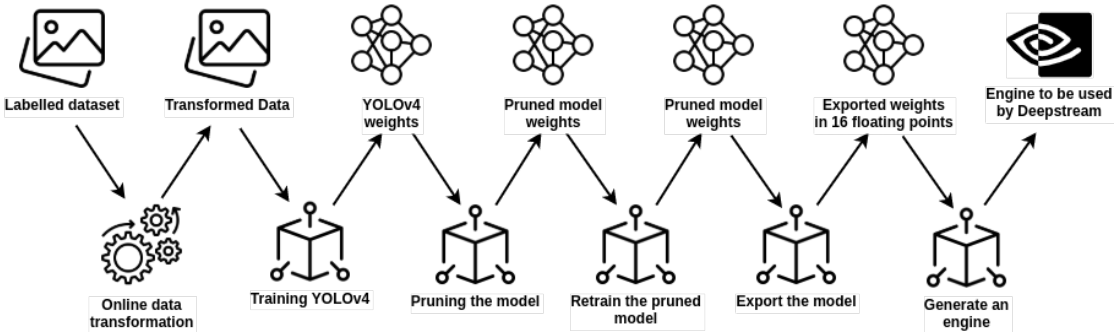


Figure 3.5: An expanded view of TLT shown in Figure 3.1

Figure 3.5 shows the tasks done by TLT, starting with the *online data transformation* we discussed previously in section 3.4. Further below, we discuss every other task, i.e. training, pruning, exporting the model and generating an engine. But first, we should explain why we chose to use TLT.

3.5.1 Why TLT and Deepstream ?

Transfer Learning Toolkit (TLT) has many advantages but also some disadvantages. In the next paragraphs, we explain each of them before justifying why we have chosen this approach over the standard one with libraries like Tensorflow, Keras or others.

Advantages:

- **Modularity :** TLT contains many deep learning models ready to be used with python notebooks. Most of them are using the same input formats which are images and KITTI labels. Since this thesis was not focused solely on the use of YOLOv4, this ability to switch between models is really useful to explore other architectures and to see how well they perform while training and on the Jetson Nano. TLT provides also pretrained backbones as well as their corresponding specification files for each architecture.
- **Simple pruning :** Since our goal is to develop an autonomous system with an embedded device that recognises insects in real-time, it means that the computing power devoted to inference is limited. Pruning is a technique used in deep learning to reduce the complexity of a model with almost no loss. To do so, TLT analyses redundant weights of the model and removes them. This process increases the inference speed as a function of the pruning ratio.
- **Optimized for Ampere Nvidia GPUs :** The Latest Nvidia GPU architecture *Ampere* has faster tensor cores that give the ability to do mixed precision training with an overall speedup of 3 times [62]. In short, it means performing as many operations as possible in the half-precision format while storing minimal information in single-precision. Training in lower precision requires less memory capacity and bandwidth as well as performs maths operations faster. Since its version 3.0, TLT can train all its models in mixed precision with a simple parameter added in the training command lines.
- **High performance on Deepstream with Jetson Nano :** Deepstream SDK, as explained in detail in subsection 3.6.1, is an inference tool that we use to perform real-time insect detection and recognition. Also developed by Nvidia, this software development kit supports TLT exported model formats which means that all object detection models will work with it directly out of the box. Unlike TLT, Deepstream is highly customizable and can perform multiple 1080p real-time inferences at the same time on an embedded device like the Jetson Nano. It is even able to track insects over the frames meaning that counting the number of different insects is possible.

Disavantages :

- **Not suitable for research purposes :** Since TLT runs on a modified instance of Docker, it is impossible to install it on a system that does not

have Docker installed. We tried for days to install a Docker alternative called Singularity on CECI clusters but without success. TLT was constantly asking for the file `docker.sock` meaning that it needs a Docker instance.
In terms of models compatibility, TLT exports its models in custom file format `.etlt` meaning that it is impossible to use them with other libraries like Keras or Tensorflow. In terms of configuration, only TLT's specification files are configurable and each parameter only has a few words explaining their function. Given the lack of information and flexibility, TLT is not suitable for research purposes.

- **Still in development phase :** TLT is in the development phase and still needs a lot of improvements related to statistics, error management and installation instructions. In terms of statistics, for some models it is unclear which formula has been used for the metrics. The user has to ask the moderators on the NVIDIA developer forum to know exactly what is happening under the hood since only simple command lines are used. In terms of errors, some are unclear or not understandable, and since its forum is certainly active but not as complete as other sites like Stack Overflow, a lack of information is often felt.

Our choice :

TLT has been made for businesses that want a computer vision pipeline working *out of the box* with restricted model customisation. From training until inference phases, TLT is efficient only if used with NVIDIA hardware and software ecosystem. We have chosen this approach over the standard ones such as Tensorflow or Keras because the goal of this thesis is to develop an entire computer vision pipeline starting from training until embedded inference and TLT is doing exactly that efficiently as long as no advanced customisation is needed. Another reason is that Deepstream is optimised for real-time embedded inference.

3.5.2 Training with TLT

Once we have made our choice concerning the model we want to train, TLT provides some command lines that launch the training. The only mandatory parameter to pass is the specification file, which is a protobuf file[1] containing every hyper-parameter divided into five categories; online augmentation, model configuration, training, evaluation and inference. When performing the training, four out of

[1] https://developers.google.com/protocol-buffers

these five are important: training, evaluation, online augmentation and model configuration.

The model configuration hyper-parameters defines the YOLOv4 architecture. We can choose which backbone to use, whether to freeze some blocks or not during training etc. The backbone has an impact on the accuracy we can expect to reach by the end of the training, for example, a ResNet34 should reach better results than a ResNet18 since it is deeper and thus offers more parameters to tune at training, but it does also lengthen considerably the duration of training.

The reason to tune the training hyper-parameters is obvious, as it directly influences how good the model will become. Well-chosen hyper-parameters have a big impact on the quality of the trained model. These hyper-parameters include the number of epochs, the batch size, the learning rate, the loss function, etc.

```
training_config {
    enable_augmentation: True
    enable_qat: False
    batch_size_per_gpu: 1
    num_epochs: 20
    pretrained_weights: "/workspace/tlt-experiments/insect-thesis/data/faster_rcnn/resnet_10.hdf5"
    output_model: "/workspace/tlt-experiments/insect-thesis/data/kitti/final/faster_rcnn/frcnn_kitti_resnet10.tlt"
    rpn_min_overlap: 0.3
    rpn_max_overlap: 0.7
    classifier_min_overlap: 0.0
    classifier_max_overlap: 0.5
```

Figure 3.6: Some parameters to illustrate the organisation of the protobuf file.

The evaluation gives us an idea of how well our model would perform when doing inference, by giving us multiple measurements of our model's accuracy. Its hyper-parameters have an impact on how the measurements are done, and thus on the apparent quality of the trained model. So, choosing these hyper-parameters will be of importance when choosing the model to keep or retrain with different training hyper-parameters. The tuning of the evaluation's hyper-parameters also helps us to choose the hyper-parameters we apply for inference, as inference and evaluation are close to sharing the same hyper-parameters.

Finally, the section on online augmentation is used to change every augmentation parameter. For example, the probability to flip an image vertically as explained in 3.4.

3.5.3 Pruning

The action of pruning removes filter channels with weights that are close to zero, to keep only those that are important. This reduces the complexity of the model, and

therefore speeds up the inference process. As for training, TLT provides a command line with multiple parameters to launch the pruning. This time, no specification file is needed as the possible parameters are not numerous (see section A.4).

3.5.4 From TLT to Deepstream

To run the inference from our insect detection model on the camera feed, we use a framework called Deepstream. To run the model on Deepstream, we need to generate a TensorRT engine. This engine is optimised for inference on Nvidia's GPUs and Nvidia's embedded systems like the Jeston Nano. TLT provides a converter to generate this engine from our model's weights.

To be able to run it smoothly on the Jeston Nano, the engine is limited to FP16, which corresponds to 16 bits floating point format instead of the 32 bits that most GPUs and CPUs can handle. This implies a minor loss of precision, but also less computation time and memory usage.

3.6 Inference on Jetson Nano with Deepstream SDK

Deepstream SDK is a multi-platform scalable inference framework focused on intelligent video analytics. It works with TLT pretrained models and is optimized for embedded devices such as Nvidia Jetson devices. In this section we explain briefly how Deepstream works, how we used it, show some results and benchmark before explaining why we have chosen this platform over the standard libraries.

3.6.1 Deepstream SDK

Deepstream overview

Deepstream is a streaming analytic toolkit to build AI-powered applications. It takes as input one or more camera, network or file streams and outputs statistics about them running in real-time. It can use all object detection algorithms that TLT uses such as Faster R-CNN, SSD, YOLOv4,... It is based on C but also provides a python script to run inference. We modified a Deepstream python

app to run real-time inference with a CSI camera and write insect detection logs. Deepstream has two configuration files: one to define Deepstream's behaviour and the other for the inference model's behaviour. For more information about Deepstream see this reference [70].

Inference pipeline :

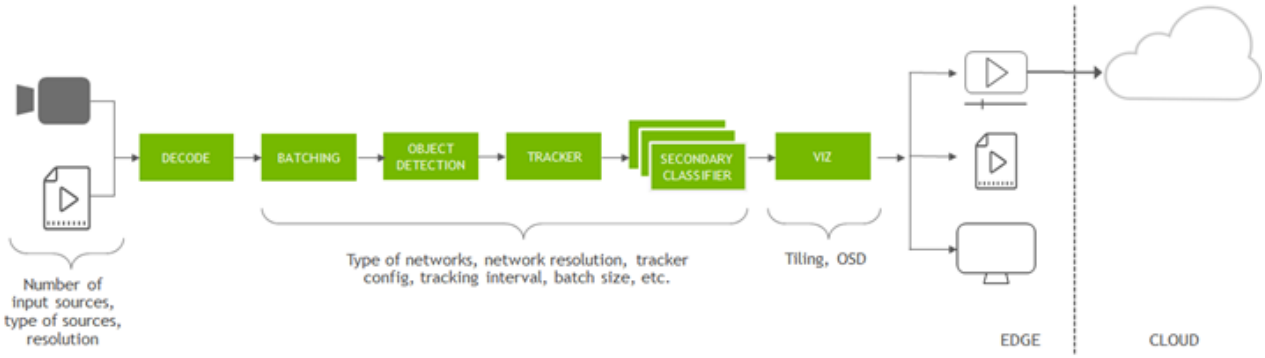


Figure 3.7: Deepstream typical pipeline [70]

First the source(s), in our case the CSI camera, is decoded then batched. Then the insect detection is performed on the batch. One key feature of Deepstream is the ability to track similar object between frames. This allows the inference pipeline to count the distinct number of insects instead of counting all the insects detected in every frame. This is done with the next element called *tracker* on the pipeline. Then, Deepstream classifies a second time the objects detected in the images. In our case, we do not use this feature but it can be a useful improvement with a more powerful embedded device. The output stream is then displayed, saved and/or streamed directly with bounding boxes predictions, class names and confidence levels.

Chapter 4

Autonomous trap

4.1 Global goal

Now that we have seen how a model can recognise an object. It is interesting to speak about the trap that runs this model and carry out the insects monitoring. First, we define its goal. In a nutshell, our system has to :

1. Take photos of nocturnal insects. We use those pictures to make a custom training set after we annotated them. These photos are taken by a python script run by the Jetson Nano, as has been explained in *subsection 3.2.2 Data collected in the field*
2. Monitoring insects attracted by the trap. As we saw in the previous chapter, a trained inference model will carry out this task. This model also generates logs containing various information about the moment of capture.

The trap should be completely autonomous, meaning it has to be possible to set it up anywhere and to run it without any involvement, except to collect the logs. That requirement implies that we must have a power source included in the trap and batteries to store the power generated. We will expand upon this later on, in section 4.3.

Despite being called a trap, the system does not restrain any insect caught. It uses a black light to attract the specimens in front of a camera and nothing more. The use of black light is a standard method used by entomologists to attract the target of our trap: nocturnal insects.

4.2 Nocturnal insects

The first step to know how this type of trap should be made is to get a better idea about what insects are nocturnal and how they live. Nocturnal insects account for a good portion of all the insect species, although it is hard to get an estimation of their exact number, as the behaviour of some species may vary during their life. Given that insects are the most diverse group of animals, it is important to get an idea of the taxonomy being used to classify them in recognisable categories, so that we may decide what our model should be able to recognise.

4.2.1 Taxonomy

Taxonomy often refers to biology taxonomy, the science of classifying all types of life forms. An enormous amount of work has been done in recent centuries by entomologists to expand it, resulting in around 6.7 million known species of animals, plants and micro-organisms, if we refer to GBIF [7]. The taxon *Insecta* alone, contains around 2.4 million species. It is therefore almost impossible to annotate by ourselves a dataset for every species. For this reason, we decided to classify insects by their orders, the highest classification taxon among insects. We made an exception for moths, for which our model can recognise two big families named Noctuidae or Geometridae. The idea is to see if our model can classify families as efficiently as it classifies orders. Or in other words, will the small differences between two families that are part of the same order be enough to make them distinguishable.

The number of insect orders varies from 29 to 34 depending on the source [18] (although GBIF provides us with a list of 43 different taxa under Insecta). Table 4.1 represents the major orders of insects in terms of species number and Figure 4.1 shows how the hierarchy of insect classification is organised. Families are often merged following the discovery of a new categorisation scheme. In the following paragraph as well as in Figure 4.2, orders of insects and moths families present in the training images of the model are briefly described.

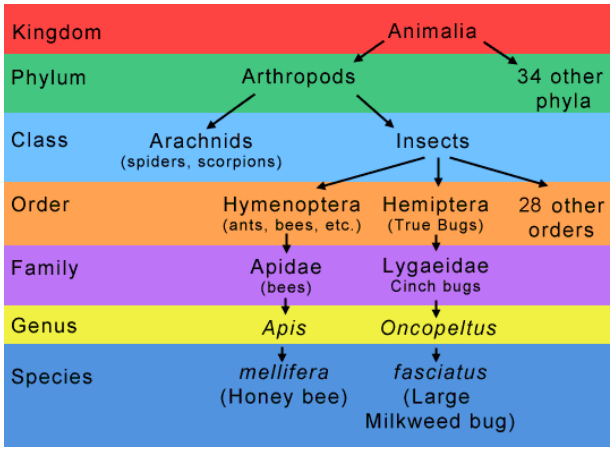


Figure 4.1: Insect classification diagram (source: [6])

- **Noctuidae :** The second-largest Lepidoptera butterflies family after the Erebidae family and counts about 1089 genera and 11272 species [9]. After observations, their main specificities are folded wings in triangle shape when idle, drab and hairy wings most of the time.
- **Geometridae :** Another big Lepidoptera family mainly recognisable by their unfolded and solid tones wings when idle.
- **Diptera :** Often called flies, this large order of around 150 thousand described species is easily distinguishable by their moving head, their large eyes and their mouthparts made for sucking or piercing.
- **Coleoptera :** Also called beetles, this order has an exoskeleton and their front wings are hidden by wing cases. The most famous one is the insect called lady beetle also known as ladybug.
- **Odonata :** This order includes dragonflies and damselflies. Their slender colourful bodies and large eyes are the main characteristics of this order.
- **Orthoptera :** Crickets, grasshoppers are part of the Orthoptera order, easily recognisable by their huge rear legs that allow them to jump high, as well as their typical sound during summer nights.
- **Hemiptera :** Much more colourful and diversified in size, Hemiptera, called *true bugs*, also have mouthparts made for sucking or piercing like the Diptera.
- **Hymenoptera :** When you see insects with yellow stripes, they probably belong to the order Hymenoptera because bees, bumblebees, hornets, wasps

all belong to it. Although many other species of this order will not carry stripes since, for example, ants are also part of it. They almost all have in common two pairs of thin membranous wings, chewing mouthparts and large compound eyes [38].

- **Trichoptera :** Closely related to Lepidoptera, this order contains around 14500 moth-like species. They are usually smaller than Lepidoptera and have two pairs of hairy membranous wings.

(a) Coleoptera (b) Geometridae (c) Hemiptera

(d) Noctuidae (e) Trichoptera (f) Odonata

(g) Orthoptera (h) Hymenoptera (i) Diptera

Figure 4.2: Orders and moths families included in the training set

Largest orders of insects				
Order Name	**Common Name**	**Identified Species**	**Characteristics**	**Description**
Coleoptera	Beetles	400,000	hard wing	Pair of hardened wings meet in the straight line down back
Lepidoptera	Butterflies, moths	150,000	scaly wings	Two pairs of large wings covered with small scales
Hymenoptera	Bees, wasps, ants (only have wings at certain times during life cycle or not at all)	130,000	membrane wings	Two pairs of thin, clear membranous wings; front pair larger than the rear
Diptera	Flies, midges, mosquitoes	120,00	two pairs of wings although the second pair is reduced in size	One pair of wings; the second pair is reduced in size and often not seen
Hemiptera	True bugs, cicadas, aphids	82,000	half wings	Triangle on back behind the head formed by the wings
Orthoptera	Grasshoppers, crickets, katydids	20,500	straight wings	Four wings, front ones thickened; jumping hind legs

Table 4.1: Orders of insects (source: [32], Moisset 2008, *Insect orders* 2014)

4.3 The layout of the trap

Now that we have established which insects should our trap capture, we can finally explain how our trap is built and why we chose the components we did. First, Figure 4.3 shows an overview of the layout, which will give you a good idea of how everything is organised and what our trap is made out of.

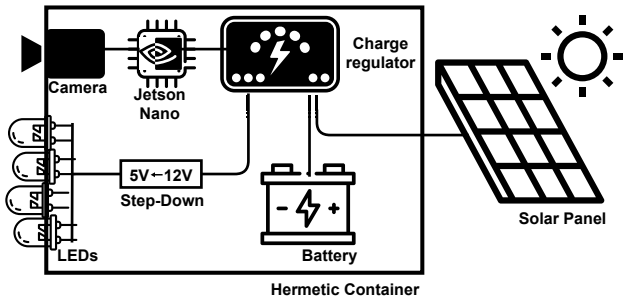


Figure 4.3: Overview of the trap layout

4.3.1 Lights and insect attraction

To make a functional trap, we have to understand how to attract insect towards it. To this end, we turn to Entomology, the branch of zoology dedicated to the study of insects. The goal of Entomology is to study not only the insect's impact on the environment, their relationship to humans and other organisms but also how they react to their environment which is what interests us here.

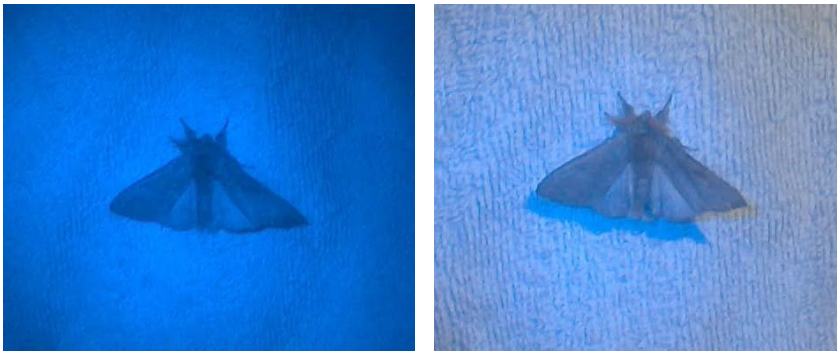
Entomologists routinely use traps to capture insects so that they may identify, classify and catalogue them. In the case of nocturnal insects, one of the most common methods seems to use a black light at night pointing toward a white sheet. One trap at a single location for multiple years will provide the data used to follow the evolution of the local insect population.

Multiple choices of black lights are available to attract insects; Mercury-vapour lamps are known to be consistently attracting the most moths thanks to their emissions in the UV spectrum, although this type of lights are now banned in the EU if used for lightning purposes and thus can not be as easily obtained. In addition to this fact, we chose to use LEDs in our system for multiple other reasons. The first being that a good selection of LEDs can be as effective as a fluorescent ultraviolet light tube, as suggested by Ryan S. Zemel and David Houghton (2017)[74], although

Diptera are an exception and seem to be less attracted. Adding to this, LEDs are better for us in practice since they often require either 5V or 12V which is easy to obtain through our charge regulator. Another advantage that cannot be disregarded in an autonomous system is that they also consume less power.

We have loosely based our choice of a light source on Benjamin Price and Ed Baker (2016)[45]. Because of this, we have four LEDs: two UV LEDs emitting a spectrum between 370 and 380nm, one blue LED corresponding to 472nm and one green LED with a wavelength of 568nm. This choice of spectrum has been made to attract most of the night-flying insects in both terrestrial and aquatic habitats since it corresponds to their visibility range. This includes all the taxa we are interested in ie. flies (Diptera), true bugs (Hemiptera), beetles (Coleoptera), caddisflies (Trichoptera), parasitic wasps (Hymenoptera), and moths (Lepidoptera)[37].

In parallel to these, we also added a strip of yellow LEDs. The role of these LEDs is not to attract specimens, as yellow light is hard to see for most night insects. Actually, yellow lights are often used in installations that do not want to attract insects, for example in agriculture to prevent the attraction of pests [22]. The yellow LEDs are useful to us because UV LEDs only provide very little visible light and so we end up with mostly green and blue light. Given that our recognition system works with a camera responsive to the visible light, the addition of yellow lights that are closer to the daylight spectrum results in clearer pictures with more colours. Also, since most of our pictures in the training set are either made during the day or under the light of a mercury lamp, the yellow LEDs give our trap's pictures a light similar to the one in the training set. Which makes recognition easier for our object detection model.



(a) Without yellow LEDs (b) With yellow LEDs

Figure 4.4: A comparison of cropped pictures captured by our trap when the yellow LEDs are set or not.

4.3.2 Jetson Nano

The main constraint that impacted our choices of components, was that our trap should be deployable in the natural environment. The first obvious step to achieve this, is to be self-sufficient power-wise. This is achieved through the use of a solar panel and a battery. It also implies to mind the power consumption of the trap whenever possible, as we did with our choice of lights.

It might seem odd that we chose to use a Jetson Nano with a power consumption of around 10W rather than a Raspberry Pi which at most will use 6.4W [43]. But it turns out, after having made comparisons between a Jetson Nano (JN) and a Raspberry Pi (RPi), that the RPi is so much slower that it would require more energy at best or would be unusable at worst. To understand why we must define the role and constraints of our embedded devices.

We want to do insect detection on the edge, meaning that inference will be run directly on our embedded device. Capturing data and sending it to a server that will run the inference is not an option, because our trap would require some access to a network which might not be possible in very remote locations.

We also have a constraint on inference time, as we want (1) to track the insect movements on the camera feed without counting the same insect twice and (2) not miss an insect that will pass briefly in front of the camera. This time constraint forces us to have real-time detection. Since inference with YOLOv4 on a RPi takes several seconds [2], we have two options to make it real-time: either we use a faster object detection model but with a lower accuracy which is not optimal, or we can capture pictures/videos at night and do inference on those during the day. But after some calculations, we realised that the second option could not only require more power since the device has to run for a longer period of time, but it was actually not feasible. If inference on an image takes two seconds then a camera feed at 30 Frames Per Second during a complete night, of say 6 hours for the sake of this example, will take around 360 hours to be processed.

$$\begin{aligned}\text{Time to process} &= \text{Video length} * \text{FPS} * \text{Inference time per image} \\ &= 6 * 30 * 2 \\ &= 360 \text{ hours}\end{aligned}$$

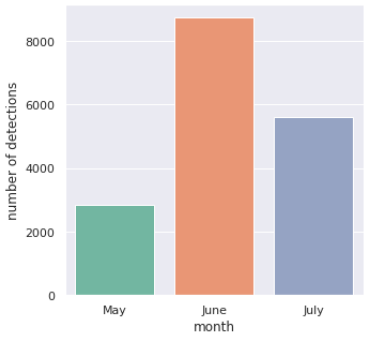
In conclusion, a JN is the better choice compared to a RPi. Mainly because it has been specifically developed to run Deep Learning models. Its computation speed makes it ideal to run inference on the edge and ends up consuming less power than a raspberry pi would. With this comes also the benefit of being compatible with Deepstream as we saw in the previous chapter.

4.4 Monitoring

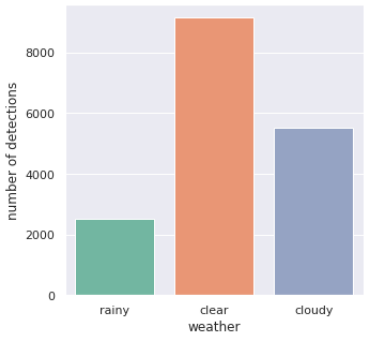
To keep track of our insect detection, a file is created at the beginning of the inference process and is fed in real-time with a buffer. This buffer is emptied after a fixed number of unique detections and saved into a log file. Before a manual shutdown, our script automatically saves remaining detections that have not yet been saved and generates several plots to visualize inference statistics. As 5.1 explains, the weather was cold and rainy for months in Belgium in January-May and thus we did not have the opportunity to fully test our inference models in-the-field. Despite this, we still wanted to show the outputs that an entomologist could expect from our autonomous system. The system can capture the following elements for each detection performed:

- The order or family of the insect
- The exact moment when it was recognised. In this case, it is expressed with the `datetime` python format that contains the year, month, day, hour, minute and second field.
- The moon phase at the time of detection. This field is important since nocturnal insects are oriented by the moon.

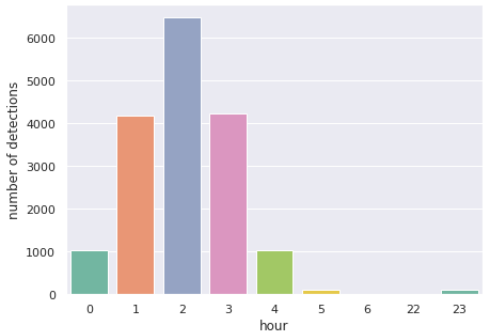
With more sensors and suggestions of modification by competent entomologists, more data can be registered at detection time. For example, a humidity sensor can be attached to GPIO pins to monitor the number of detections in function of the humidity in the air. A temperature sensor can also provide interesting results. We generated artificially some detection data with a python script to show some graphs that can be produced with the inference logs. The data are fake and are generated with random or probabilistic rules. Here are some examples of graphs we can generate :



(a) Detections per month

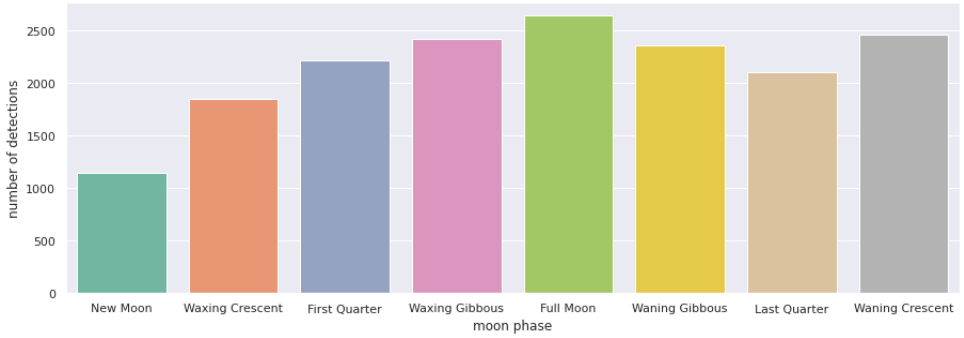


(b) Number of detections in function of the weather

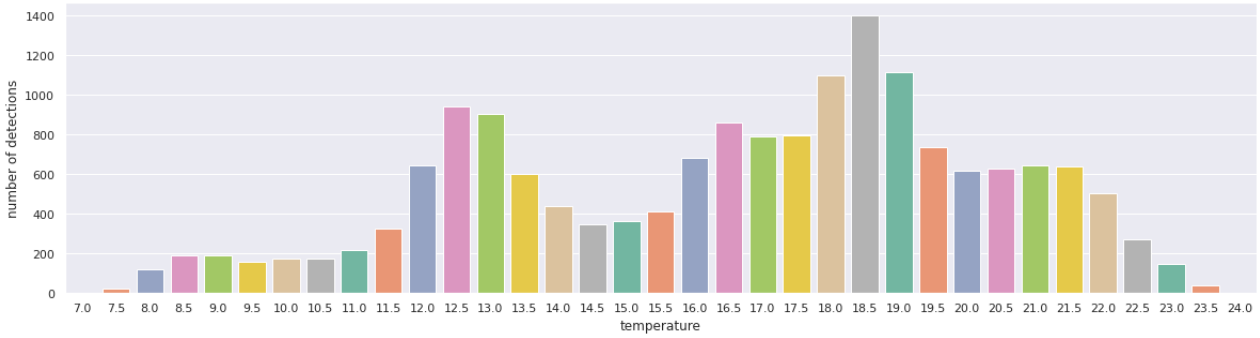


(c) Number of detections at each hour of the night

Figure 4.5: Statistics on detections



(a) Detections per moon phase



(b) Number of detections in function of the temperature

Figure 4.6: A picture of our trap in action, attracting insects on a white drape.

Chapter 5

Results

5.1 About the weather

It may be an unusual way to start our chapter on results, but considering the impact the weather had on our work there are good reasons to talk about it. April 2021 has been the coldest month of April in 35 years [19] and affected the insect population. Insects start to move although very slowly at around 7°C and reproduce themselves at around 22°C [64]. This year, we had nights with temperatures under 7°C up until late May, which means few insects were present (except a few immobile Coleoptera). We did have some success in our observations during the few warmer nights, but we are still far from the number of pictures needed to be usable as a training or even testing set.

What does this entail for our results ? First and foremost, the metrics do not show how well our trap can recognise insects in the field but only how well it can recognise pictures of insects taken from GBIF, since that is where pictures of our test set come from. If we were able to use pictures taken from our trap, our results would probably be lower than they are now. This is entirely normal, as much as augmentation can help to bridge the gap between the two different data sets, our model will never be as good as one that would have been trained solely on pictures coming from our trap.

(a) A Calliteara pudibunda on GBIF

(b) The same insect seen from our trap

5.2 Object detection performance evaluation

To assess the performance of an object detection algorithm, many metrics exist. These metrics as well as their datasets have been created for competitions of object detection algorithms. Usually, to assess a new algorithm, scientists use the evaluation assessment tool provided by the dataset itself. For example, PASCAL VOC datasets [61] use the PASCAL VOC annotation format which has a MATLAB evaluation code available. To compare different algorithms, it is essential to use the same metrics, otherwise, the comparison is biased. Below, we present the metrics used in order to calculate the PASCAL VOC metric.

5.2.1 Intersection Over Union (IOU)

Intersection-over-Union (IoU), as explained in the glossary and shown in Figure 5.1, is an evaluation metric used to evaluate the overlapping area between the ground-truth bounding box $BB_{g.t}$ and the predicted bounding box BB_p. If the IoU is greater than a threshold chosen by the metric and both boxes have the same label, then the detection is considered valid (True positive). Otherwise, it is considered a False positive.

- True positive (TP) : valid detection, the prediction bounding box overlaps the ground-truth bounding box ($IOU \geq threshold$).

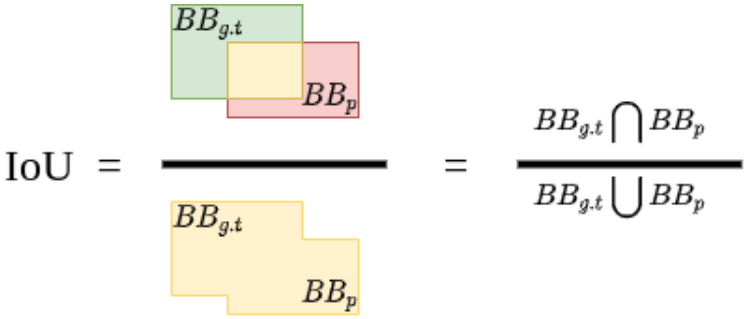


Figure 5.1: Intersection over union between a ground-truth bounding box ($BB_{g.t}$) and a prediction bounding box (BB_p)

.

- False positive (FP) : Invalid detection, the prediction bounding box does not overlap the ground-truth bounding box ($IOU < threshold$).
- False negative (FN) : Ground-truth bounding box is not detected by the algorithm.
- True negative (TN) : In object detection, this metric does not apply. It represents all the bounding boxes that should not be detected and that have not been detected.

Usually, the threshold is set to 0.5 but it varies depending on the competition and its dataset.

5.2.2 Precision

Mathematically, it is the ratio of the number of true positives to the number of positive predictions. In other words, *it is the ability of a model to identify only relevant objects* [40] and is given by :

$$\textbf{Precision} = \frac{TP}{TP + FP} = \frac{TP}{\#BB_p}$$

5.2.3 Recall

Recall is the ratio of true positives to the number of ground-truth bounding boxes. In other words, *it is the ability of the model to find all the ground-truth bounding*

boxes [40] and is given by :

$$\textbf{Recall} = \frac{TP}{TP + FN} = \frac{TP}{\#BB_{g.t}}$$

5.2.4 Precision x Recall curve

The precision x recall curve is a notorious way to evaluate the performance of an object detector algorithm. This curve is plotted for each class individually and shows how the precision reacts when the recall increases and the confidence threshold changes. If the precision stays high when the recall increases, it means that a higher confidence threshold during inference still gives high precision and recall values and therefore the predictions have larger intersections with ground-truth bounding box. Another way to evaluate an object detector is to look at its predictions. The ideal goal of an object detector is to have both high recall and high precision, which means that it largely predicts ground-truth boxes with few false positives. On the other hand, a poor object detector will increase the number of false positives in order to achieve a high recall and therefore the slope of the curve will directly decrease.

5.2.5 Average precision

Another way to interpret the graph is by calculating the area under the curve precision x recall for each class. This area is called *average precision (AP)* and is ranged between 0 and 1. Theoretically, the average precision is the precision averaged across all recall values between 0 and 1 [36].

$$\int_0^1 \rho(r)\, dr$$

where $\rho(r)$ is the precision at recall r. To approximate this area, we can use interpolation. As we explained above, formulas to compute metrics can change between competitions but also over time. For example, the PASCAL VOC challenge first used *11-point interpolation* to estimate the average precision area. From 2010, the challenge replaced the *11-point interpolation* by an *all points interpolation* method.

5.2.6 All points interpolation

Instead of using 11 equally spaced points for interpolation, this method interpolates at each recall level to obtain the area under the curve. The formula of this interpolation is the following [41] :

$$\sum_{n=0}(r_{n+1} - r_n)\rho_{interp}(r_{n+1})$$

with

$$\rho_{interp} = \max_{r_\lambda : r_\lambda \geq r_{n+1}} \rho(r_\lambda)$$

where r_λ is the recall at point λ, $\rho(r_\lambda)$ is the measured precision at recall r_λ and n is the number of recall points. This formula is taken from "*A survey on Performance Metrics for Object-Detection algorithms*" (2020) [41], the original paper that introduces the performance evaluation tool that we use in the following sections. As we can see on Figure 5.2, the precision begins at 1, then rapidly goes down and forms wiggles until the end of the recall axis (see blue line Figure 5.2). To compute this area and reduce the impact of wiggles of the curve caused by the variation of the threshold, the AP is slightly overestimated by taking the maximum precision value to the right of the current recall r (see red dotted line Figure 5.2 [41]. Once the AP of every class is found, we can calculate the mean to obtain the mAP. The mAP is a popular metric to measure the accuracy of object detection models.

5.3 Comparisons

For the results presented below, we have used the following workstation :

- **CPU :** Intel I5-10400F 6-cores 12 threads @ 4GHz.
- **GPU :** RTX 3060Ti Msi Ventus 3X OC 8GB VRAM (*Ampere* architecture).
- **RAM :** 2x8GB @ 2666MHz.
- **Storage :** SSD SATA (OS) and HDD @ 5400 rpm (data)
- **OS :** Ubuntu 20.04 LTS

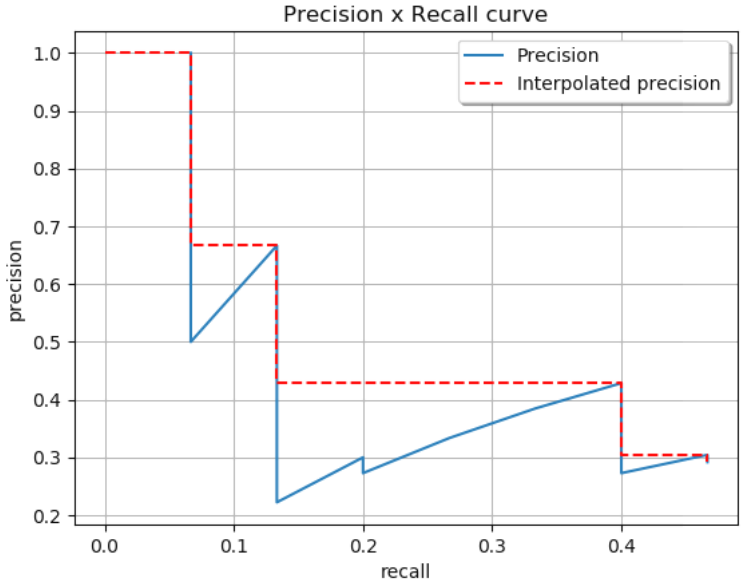


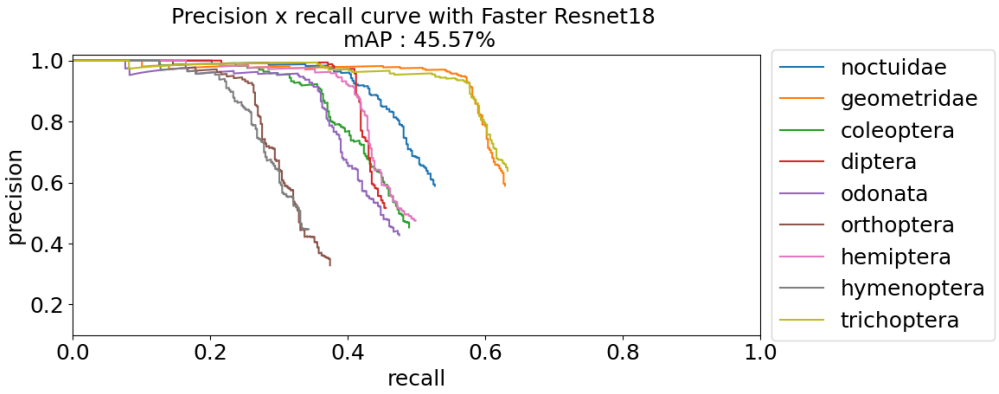
Figure 5.2: All points interpolation of the precision x recall curve [41].

For the parameters used in benchmarks below, we used an Intersection-over-Union of 0.5, 20 epochs for each model, and various learning rates depending on the algorithm used. We also used the *soft-start method* to start slowly the learning in early epochs, reach the maximum learning rate then slowly decrease the learning rate in the later epochs. This allows the accuracy to be refined before the end of the training, and therefore better results to be obtained in certain cases. The other parameters have not been modified and are those proposed by TLT, allowing us to evaluate their performance straight out of the box.

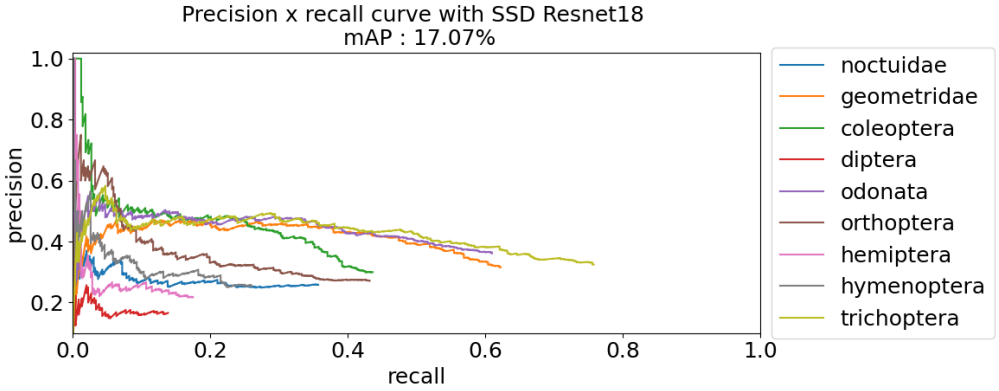
5.3.1 SSD vs YOLOv4 vs Faster-RCNN

In this section, we compare multiple object detection algorithms with ResNet18 backbones on our test set of 4263 images from GBIF. As we can see in Figure 5.3, all three have significant performance differences.

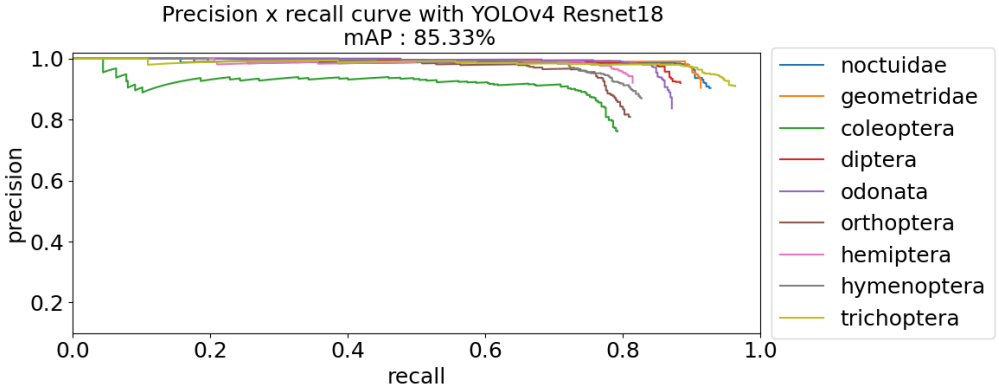
First, we can see that the Single Shot Mutlibox Detector (SSD) barely reaches 20% of ground-truth boxes detection, with a precision of around 0.4. This tells us that more than half of the predictions are incorrect and some ground-truth boxes are impossible to detect for SSD. We also see that SSD detects some classes better than some others, which means that some fine-tuning and augmentation are needed. As a result, low classes curves in SSD directly impact the final mean



(a) Faster with Resnet18



(b) SSD with Resnet18



(c) YOLOv4 with Resnet18

Figure 5.3: precision x recall curves for Faster, SSD and YOLOv4

average precision (mAP) and score a low 17.07%.

Faster R-CNN on the other hand is much better, reaching an mAP of 45.54%. Training Faster R-CNN with a larger backbone such as ResNet34 gave us better results with an mAP of 68.99%, still meaning that further experiences are needed to fine-tune it correctly. Class differences are still there since some classes end up with less than 40% of ground-truth boxes recognised while others have more than 60% and with a precision far superior to SSD.

YOLOv4 exceeded our expectations with a stunning mAP of 85.31% without fine-tuning which matches results as proposed by Ramalingam et al. [47]. With a strong average precision of 0.95 until a recall of 0.8, except for the Coleoptera class, YOLOv4 delivers a strong object detection experience. This is the reason why we have chosen this algorithm over Faster R-CNN and SSD.

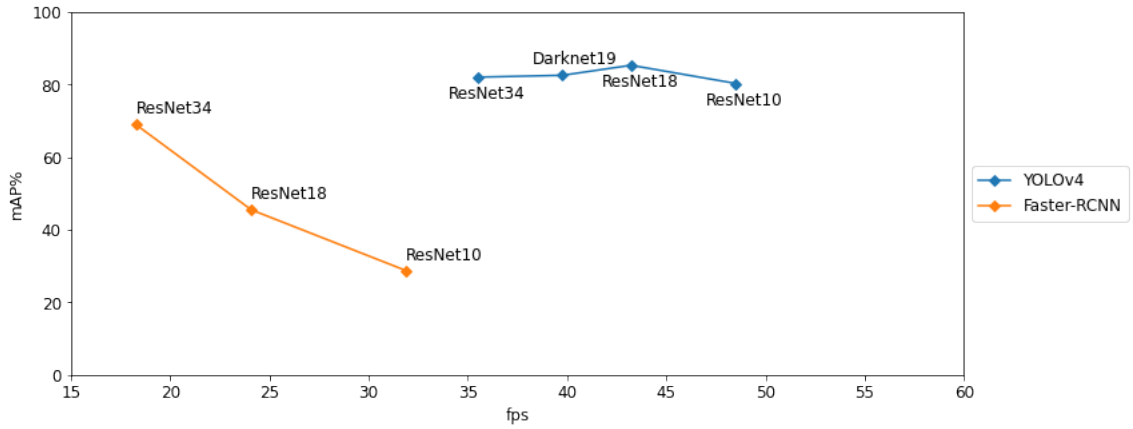


Figure 5.4: Comparison of the mAP and inference speed of Faster R-CNN and YOLOv4 with different backbones

Finally on Figure 5.4, we compare the speed and accuracy of Faster R-CNN and YOLOv4 with different backbones. As we saw, YOLOv4 delivers better results both in speed and accuracy and that even with a shallow backbone. Surprisingly, YOLO does not seem to be much impacted by its backbone depth. This probably changes with more fine-tuning and/or more epochs during training as we can not think of any reason why YOLOv4 would not benefit from deeper backbones.

5.3.2 Online augmentation influence on YOLOv4

To understand the importance of augmentations and their influence on accuracy, we trained a model after disabling each of them. This model has an mAP of 81.51%, which, compared to the 85.33% of our model, is not negligible although not very impressive.

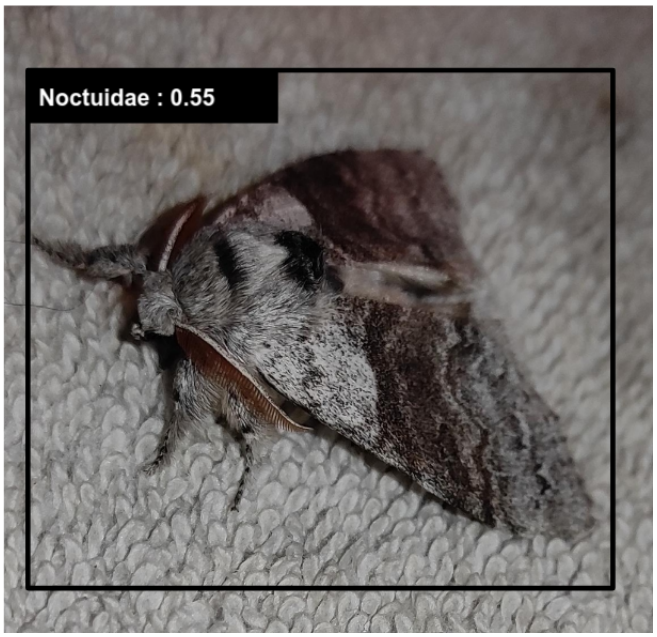
The biggest change happens when we run the inference on images taken from our trap. It makes sense since we chose the augmentation to better represent what our trap might catch. Since we do not have enough images to make a good test set, we can not evaluate its impact. But only by looking manually at the output after inference of the few captures we did make, we can tell the positive impact that augmentation has (see Figure 5.5).

With augmentation, the moth gets recognised and a box is drawn correctly giving its position. Without augmentation, two classes are assigned; one is completely false as the moth is recognised as a Hemiptera and the second is right but its position is completely wrong as a huge box covering more than half the picture is drawn. We chose this picture as an example to show how the model has a hard time both locating and recognising the insect. It should be noted that this particular moth is not actually a Noctuidae, but part of an order not recognised by our trap, while bearing the most similarities with a Noctuidae. Sadly, our choice of pictures for this example is limited.

5.3.3 Backbones depth benchmarks

As shown by Figure 5.6, backbone depth slightly impacts the mAP while impacting a lot the average precision of individual classes. Indeed, we can see that ResNet10 is relatively stable but has problems with high confidence images of some insect classes.

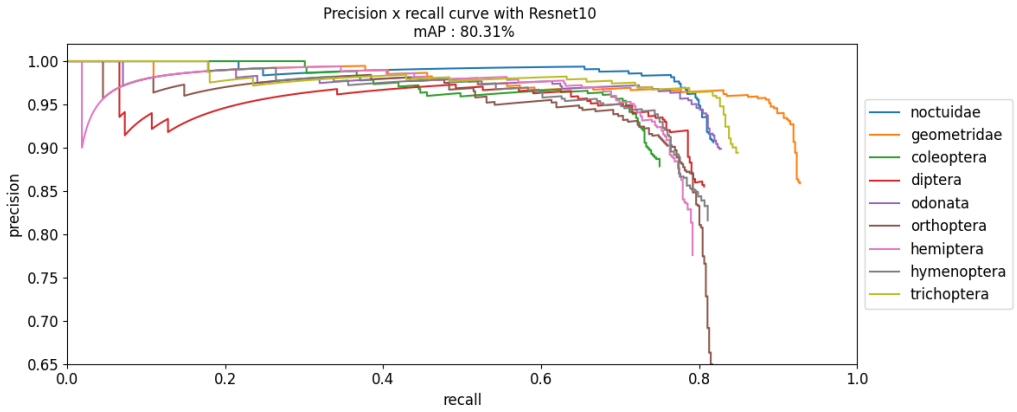
The precision with a recall between 0 and 0.2 quickly drops to 0.9 meaning that some high confidence images are false positives, especially for Hemiptera and Diptera. This is interesting since both classes are physically quite different. As a reminder, Diptera are flies and are often black while Hemiptera are often brightly coloured. Once the recall is between 0.75 and 0.92, most classes' precisions decrease when YOLOv4 tries to detect all the ground truth boxes.



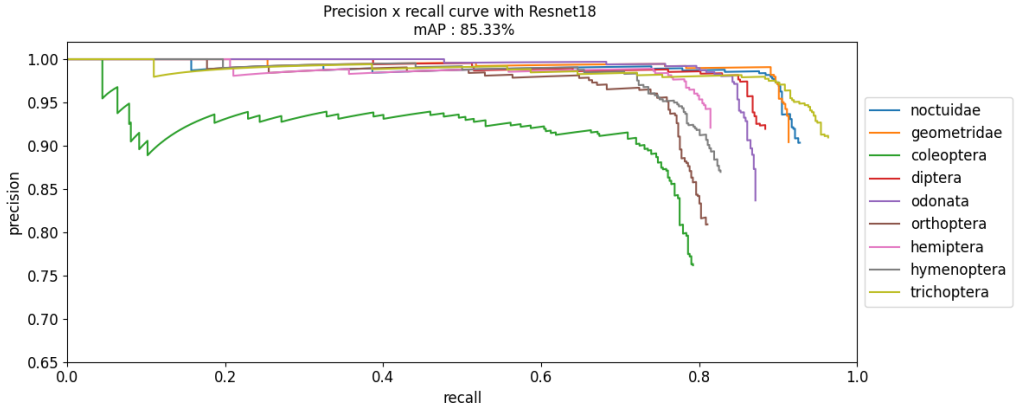
(a) With augmentation

(b) Without augmentation

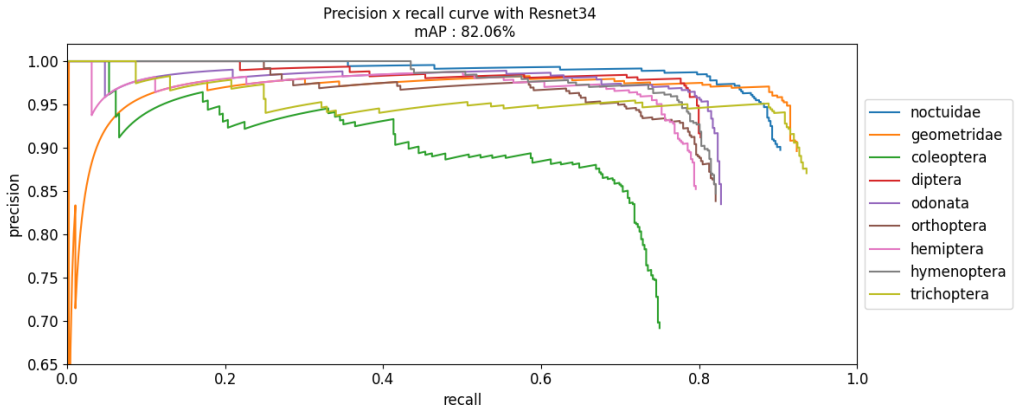
Figure 5.5: Two images showing the result of the inference of a model with and without augmentation. These images are cropped from the originals that are too large to appear here (4640x3472 pixels).



(a) Resnet10



(b) Resnet18



(c) Resnet34

Figure 5.6: YOLOv4 precision x recall curve with various Resnet backbones

ResNet18 on the other hand is really stable overall up to a recall value of at least 0.8, although it seems to have trouble detecting Coleoptera. This result can be explained by looking at the inference output on the test set (see Figure 5.7). Most of the images are similar to picture (a). The detection is accurate with high confidence and can even recognise blurry insects in the background, but sometimes YOLOv4 detects colourful round objects or parts of insects as Coleoptera like in images (b) and (c). This is because a lot of the Coleoptera are ladybirds (ladybugs) in our training set, and are both round and varied in colours. However, since our trap is deployed in front of an uniform white background, this problem should not arise during our in-the-field detection. On image (d) we can see that the detection is still quite accurate even if the Coleoptera curve is lower than other curves.

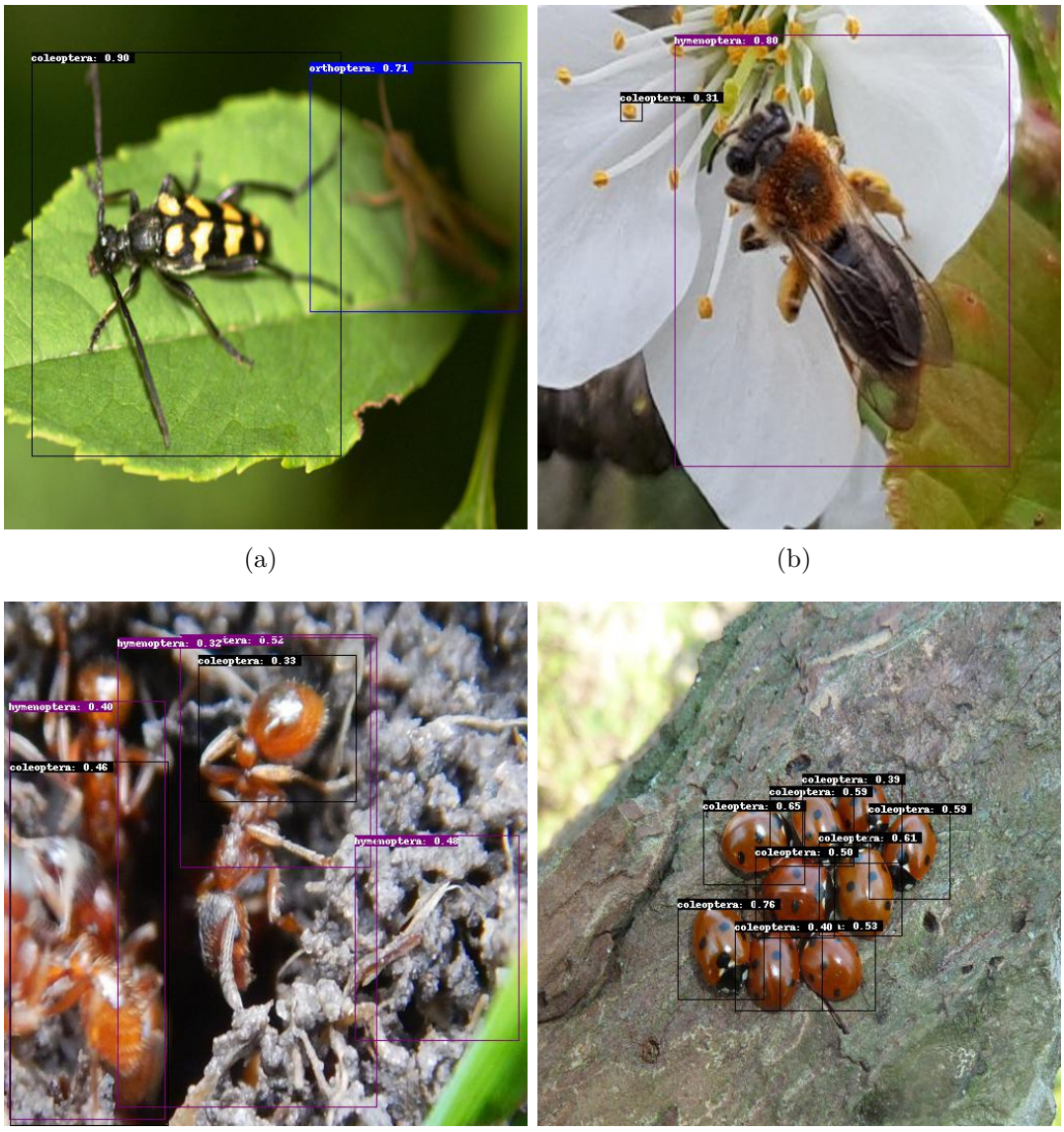
With ResNet34, Coleoptera get an even lower result than with ResNet18, meaning that the backbone depth has a negative impact on Coleoptera detection. It also creates noise until the recall reaches 0.5 in high confidence images.

Overall all three are performing quite well with a mean average precision of 80.31%, 85.33% and 82.06% for ResNet10, ResNet18 and ResNet34 respectively. We have chosen ResNet18 because it has a stable precision of 1 until a recall value of 0.8 for most of the classes. Adding some augmentation to the Coleoptera and diminishing the over-representation of ladybirds amongst them could maybe fix their "low" curve.

5.3.4 Comparison size of training set per class

Our dataset contains about 5000 images per class. This is something that is far from common if we are to believe all the recommendations that can be seen online. They suggest using around 1000 image per class. Supposedly, the origin of this recommendation comes from the original ImageNet classification challenge. A thousand images were enough to train the image classifiers back then, and so it became somewhat of a standard [69]. This can also be observed in publications about insect detection, as for example is proposed by Ramalingam et al. (2020) [47], which uses 1000 images per insect class. And with the addition of transfer learning, it seems that a number as low as 150 to 500 delivers good results for ResNet classifiers [55].

The reason why we used so many images is that we want to detect and classify insects by taxonomic orders. However, the orders of insects are very varied. Two insects of the same order might look completely different and might even be more similar to insects in another order. For this reason, we had the intuition that a



(a) (b)

(c) (d)

Figure 5.7: Coleoptera detection with YOLOv4 & ResNet18

thousand images per class might not be enough.

To test this assumption, we trained our model with datasets of multiple sizes. We used 6 different sets of images per class: 500, 1000, 2000, 3000, 4000 and 5000. The evolution of the mAP of each trained model is shown in Figure 5.8.

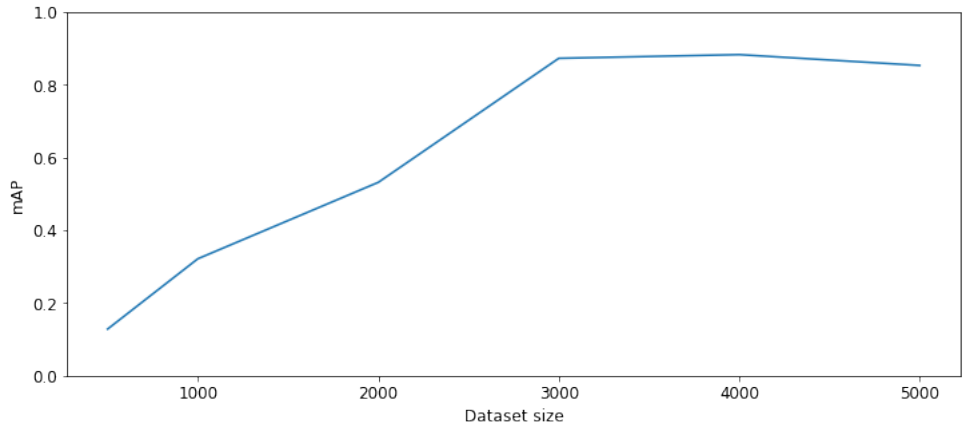


Figure 5.8: Influence on the mAP of the number of images per class in the dataset

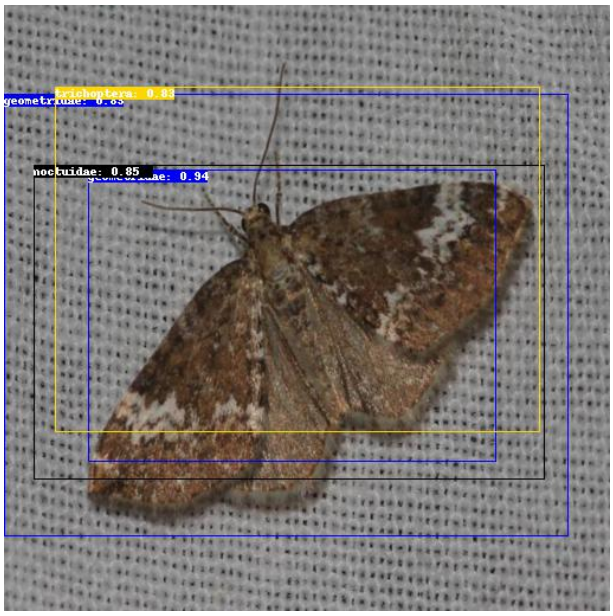
We can see that despite our use of transfer learning, 500 images per class is far from delivering good performance, with an mAP of not even 13%. This is due to two reasons: (1) multiple insects are not recognised at all, meaning we have a low recall (2) the model has a hard time distinguishing between the different insects which leads to a low precision (Figure 5.9).

Then the mAP is increasing almost linearly up to 3000 images where it starts to plateau. This seems to confirm our assumptions, more images are beneficial to distinguish insects.

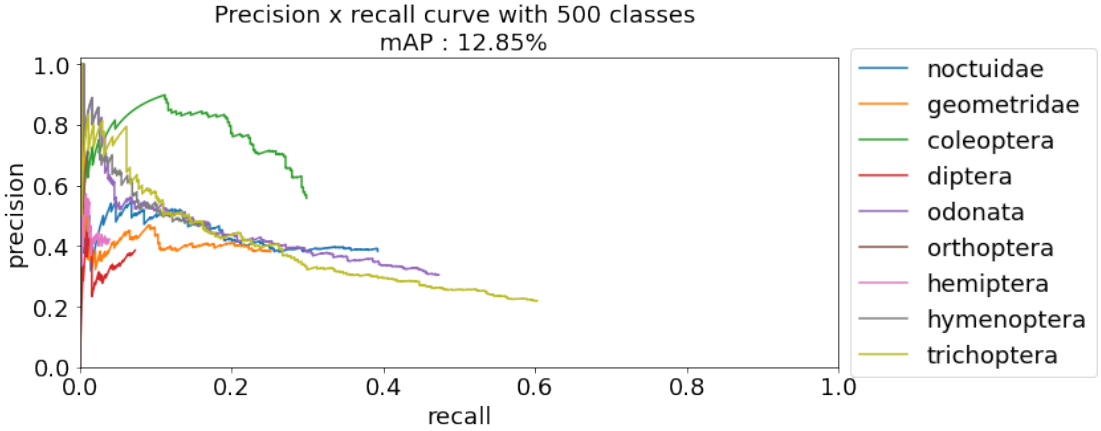
A slight increase can be observed when we reach 4000 images (from 87,3% to 88.3%). But we do not think the change is big enough to justify the use of a thousand more images.

We can also observe a decrease at 5000 images (dropping down to 85.3%). However, we do not deduce from there that an overly large dataset will always be detrimental to the mAP. It is our opinion that this is a purely incidental result, as an increase of the number of data should not induce overfitting or any other problem as far as we are aware.

What may be happening is that some species inside an order are pictured more often than others. When we add further images, these species get even more



(a)



(b)

Figure 5.9: With only 500 images per class, the model assigns multiple class to a single insect with quite high confidence. This results in a hard drop of the precision on the graph early on.

over-represented and then there could be overfitting happening on those species, as the model gets trained to recognise this species specifically. But this is only speculation because we could not find another explanation. Regretfully, we do not have the knowledge to distinguish the species by ourselves and thus we can not observe if this statement is true for our dataset.

In conclusion, around 3000 images per class seem enough to train our model. However, this is not set in stone, a deeper neural network for example might benefit from a larger number of images per class.

5.3.5 Differentiating orders and families

As said in chapter 4, we have seven orders of insects and two families of moth. These families share many similarities as well as they have their differences. We wanted to see if the model would be able to find these differences easily or not. To analyse this, we can look again at the precision/recall graph of our model (see Figure 5.10).

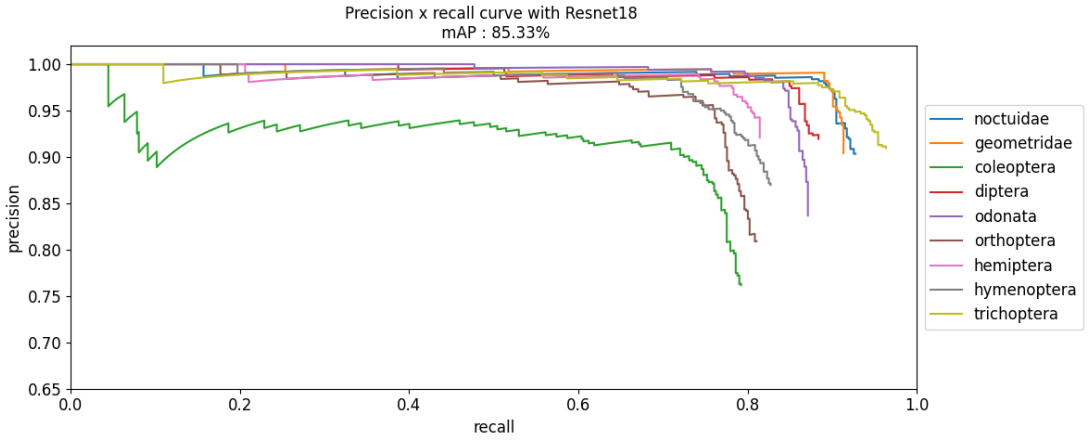


Figure 5.10

We can see that, as far as the detection is concerned, the Noctuidae and Geometridae are the second and third best classes after the Trichoptera. According to this, the model has no problem making the distinction between moths. It might even be better trained to recognise similar insects, since the Trichoptera also share some similar features with the moths and is the class with the highest accuracy. Thus, similarities between classes do not necessarily have a negative impact on classification, at least between two similar taxon families.

Chapter 6

Conclusion

In response to the diminishing abundance of insects and to collect data on their falling population, we introduced a new method of insect monitoring using object detection on the edge.

After exploration of multiple object detection architectures, our choice has been set on YOLOv4 as it is one of the top object detection models accuracy-wise and has the added advantage of being fast, attaining real-time inference speed of 5 FPS in average. We then presented the methods used to train the model and our different choices of implementation, as well as the recognition pipeline that is used in our solution.

An autonomous system deployable in remote locations has been introduced to run the object detection inference and collect data of the local nocturnal insect's population. Since the automated traps available on the market or presented in insect monitoring publications are connected to a network and the inference done on a server, we wanted to show that real-time insect monitoring on the edge is possible.

With our YOLOv4 model, we reach an mAP of 85.33% on images taken from GBIF. The model also has good results with in-the-field recognition, at least on the small insects sample we could attract. This small number of observation is due to the weather being too cold for nocturnal insects to be active.

There are multiple small but time consuming improvements we could still make to improve the accuracy of our model. Improvements like trying other models, performing more fine-tuning, etc. But the accuracy of our model is not everything,

there are other areas of improvements we could address as well.

There are some works that we had initially planned for this thesis, but were prevented to by the cold weather. First of all, we wanted to train a model based only on pictures taken by our trap. We would have trained a first model on images taken from GBIF, just like we did, but then we would have used the principle of transfer learning to train our final model solely on images from the trap. This way, we would have a model specifically made for our trap with a minimum of in-the-field pictures required. The challenge resides in finding enough specimens of each order in-the-field. Around 500 per class could be enough, but it will have to be tried experimentally to know for sure.

Another measure we would have liked to do, is observing the discrepancy between the number of insects counted by our trap versus how many were actually present. This is not a problem when using object detection on a still image since the model does not give many false positive. Especially on pictures taken by our trap since the background is a white drape on every image, so the model would not be able to confuse something in the background for an insect. The problem arises when processing the feed of the camera; here the tracker might miss an insect for a few frames and then count it a second time. This is due to our current tracking system that uses the object detection model to know where the objects are. So, whenever an object is not recognised for a few frames, which can happen if the insect moves a lot, it will be counted as a different object when recognised again. We could implement a tracking system independent from the model which would track the objects by the difference in pixels from one frame to the next, and if the position of an object corresponds to the position of a recognised insect then it would be added to the log.

Until now, we have discussed what we would do to improve our current system. But it would be interesting to build different models in the same vein as ours. For example, since we know from subsection 5.3.5 that YOLOv4 can make the distinction between similar looking insects, a model could be made to distinguish insects at the species level. Maybe even up to a distinction between the sexes as with some species, males and females can be quite different. We believe such a model can be made even though it would contain a huge number of different classes, but as YOLO9000 demonstrated, the YOLO architecture can work with a lot of classes (9000+ in this case)[48].

During our work, we had the opportunity to have a very constructive talk with an entomologist, Claire Stulemeijer to review our plans and learn more about entomology in general. But it would be interesting to go further and remain in

direct collaboration with entomologists to know if our work is aligned with their needs, also to get their input on the elaboration of an effective trap design. There is also the possibility to add modules to the trap if some particular data should happen to be especially useful to the entomologists.

To conclude, our solution proves that insect detection on the edge is possible in field conditions. We showed that insect detection using deep learning models can be done on embedded devices, removing the need for cloud computing. To run this device in the wild, we introduced a solar-powered trap that monitors in real-time local insect biodiversity.

Appendix A

User Manual

Our modifications of the annotate-to-KITTI repository[26] as well as our custom script for annotation verification are described in this appendix.

A.1 Annotate-to-KITTI fork

Here are the modes we implemented in our fork [3] of the annotation tool *annotate-to-KITTI*.

Simple annotation : The main reason why we chose to fork `Annotate-to-KITTI` is that the annotation process is fast and simple and the script straightforward. A simple drag and drop on the image displayed creates an overlay representing the newly created ground-truth bounding box. When all the objects have been correctly labelled, press "q" to move to the next picture.

Paths and folder creation A file and folder hierarchy is very important as it must be possible to sort files by their utilisation. We organised files in our own way (see Figure A.4). A dataset folder is composed of three sub-folders :

- `inference` folder contains images that haven't been annotated yet. For example, files that have been downloaded by the `gbif_parser.py` script will be automatically saved in this folder.
- `train` folder contains images that have been annotated as well as their

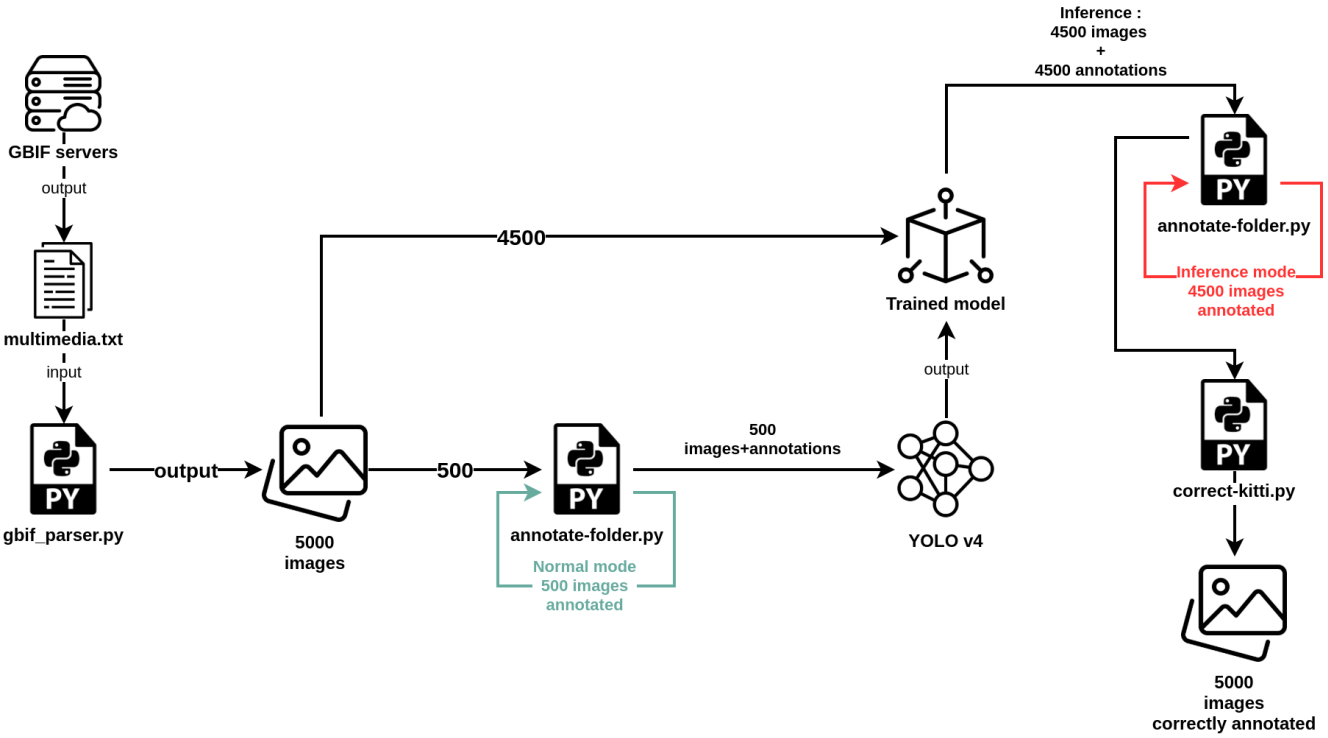


Figure A.1: Annotation workflow for each class

corresponding labels containing ground-truth bounding boxes.

- `faster_rcnn` folder contains images that have been annotated by a model via an inference process but still have to be reviewed by the user with inference mode.

To manipulate this folder hierarchy, we implemented 3 modes in the tool as shown in Figure A.2, in order to either annotate new images in the inference folder, visualise what was previously done in the train folder or review inference results.

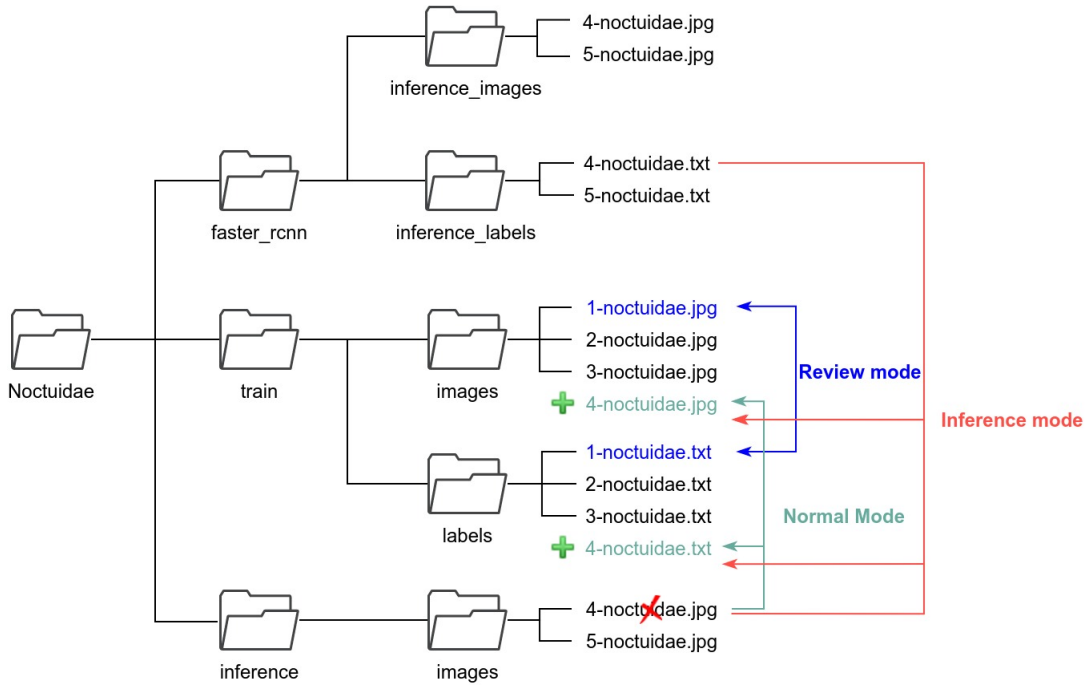


Figure A.2: Annotate-to-KITTI modes

Modes

- `Normal mode` : This mode is used to annotate new images of the inference folder and to move them to the train folder. We first used this mode when we created annotations for a new species before training the model for inference.
- `Review mode` : This mode verifies the presence of annotations already made by the user. Thus, it checks only the files present in the train folder.
- `Inference mode` : This mode reviews inference images that have been produced by the model trained with 500 images as explained in figure Figure A.1.

Each folder inside the `kitti` folder is considered as a dataset and can contain one or more classes depending on whether it is a dataset used for training the detection model or simply a folder used for annotation purposes.

Added keys :

The original author had already implemented several actions in order to quickly interact with the annotation tool such as changing the label during annotation with key "l", removing the last ground-truth box of the current image with key "c", skipping the current picture with key "n" and quitting the program with "Esc". But some actions that we think essential for fast annotation were missing, like reviewing annotations and deleting bad pictures. Here are the features we added to the script :

- **Deletion of an Image and its labels:** The current image can now be deleted with key "d". This mode allowed us to quickly delete bad quality photos that were not useful for the training process without using the file manager of our computer.
- **Previous image :** If an image has been skipped or incorrectly labelled, the user can now come back to the previous image with key "r". This allowed us to quickly review already labelled images by our inference model without restarting the program if we missed an image that was incorrectly labelled.

A.2 Correct-kitti script

Even if our tool `annotate-folder.py` allows us to quickly return to previous annotations or to delete files, we found around 20 bad annotations over more than 48000 annotations. These errors can degrade severely the performance of the trained model since learning on bad annotations disrupts the learning process. Most of the bad annotations were bad label format, badly written labels or microscopic ground truth bounding boxes. To find these errors, we passed each annotation file as input to our script `correct-kitti.py`. In the following paragraphs, we will explain how each error is handled.

- **Bad label format :** The KITTI format produced by inference and the one used in training do not have the same format. Faster-RCNN inference pro-

duces a confidence percentage per predicted bounding box. To use inference results again in training, we need to remove these confidence percentages.

- **Badly written labels :** Sometimes, multiple orders of insects can be present in the same image. In this case, we use the "l" key as explained in section A.1 to type the next bounding box label. But after having done thousands of annotations, it can happen that the user misspells a class name. This verification prevents that by using a dictionary of all the class names.
- **Microscopic bounding boxes :** To create a label with our annotation tool, the user simply has to drag and drop over the image to create a squared overlay. The problem is that a simple left-click without moving the cursor creates a small rectangle. Even if TLT has a parameter in the configuration files that prevents this kind of error from being taken into account, we preferred to solve the problem from its origin by checking the size of each annotation.

  Thanks to this script, we can be sure that every little mistake that is easily corrected by a computer has been caught. This script is easily modifiable so that it is quite simple to add in further tests, should this be necessary in the future.

A.3 GBIF parser

In order to download images from the online database explained in subsection 3.2.1, we implemented a script that downloads, converts, resizes and verifies each image in the list of URLs. This script is named `gbif-parser.py`. Here is how to use it:

```
python3 gbif-parser.py name path/to/txt/mediafile n [width] [height]
```

where :

- `name` is the name of the species
- `path/to/txt/multimedia` is the path to the file `multimedia.txt` downloaded on the gbif website.
- `n` is the number of images that the user wants to download. For our dataset, we have chosen 5500.

- `width` & `height` are the desired width and height of all the images downloaded through this tool. The aspect ratio is not to be kept because we don't want to lose elements of the image by cropping them. We decided to not resize them after their annotation because resizing labels is a long process. These parameters are optional but the recommended value is 512 since it must be divisible by 32.

Here are the actions of the script:

1. Parse the parameters
2. Create appropriate folders to store images according to the name passed as parameters
3. Parse the multimedia file to only keep URLs
4. Create a pool of threads that will share the work among them.
5. For each thread :
   (a) Download the image
   (b) Check the format of the image, if it is not `jpg`, convert and save it into it. We have chosen `jpg` since it is the default format of GBIF and it is more compressed than `png`.
   (c) Check the dimensions of the image, otherwise delete it.
   (d) Resize the image to the size passed as a parameter if this parameter is present.

Since this process can be parallelized, we decided to use threads in order to not waste time waiting for images to download, convert and resize. Here is a graph showing that the script is well parallelized (see Figure A.3) as time decreases when the number of threads increases. The only things in the pipeline that can't be parallelized are the internet connection as well as the access to the storage unit. Apparently, these potential bottlenecks do not influence the computation speed even though this benchmark was done with a 6 core 12 threads processor combined with a 5400rpm hard drive. If we had done these benchmarks with much larger images in memory, we would probably have had different results.

To keep an eye on the progress of the download, we used the functions `tqdm` combined with `istarmap`, a `starmap` version of `imap` implemented by the community of Stack Overflow [46]. Images are saved at path `data/kitti/name/inference/images` because they do not have labels yet.

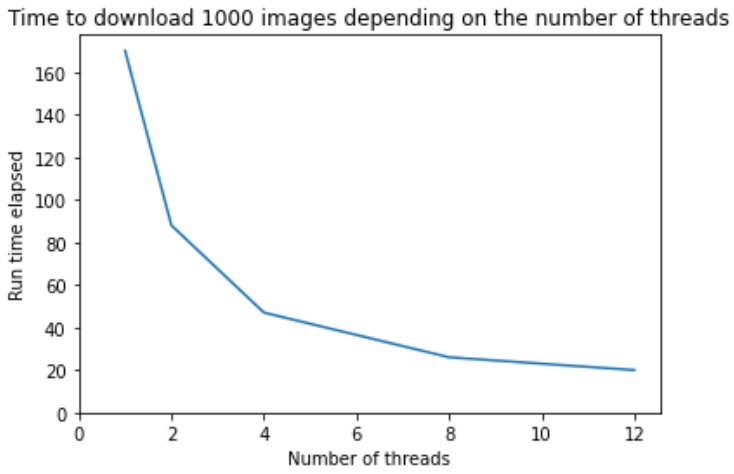


Figure A.3: Time to download 1000 images depending on the number of threads.

Figure A.4: Hierarchy of our git repository

A.4 Documentations

The documentation of TLT can be found at https://docs.nvidia.com/metropolis/TLT/tlt-user-guide/index.html. Our GitHub repository can be found at [3]. Its structure is shown on Figure A.4.

Appendix B

Components

We have mentioned some of the components needed to make our trap, but we have yet to give a complete list. Below, you will find the list of all the elements:

- A 12V 30W solar panel.
- A solar charge regulator, working with tensions of 12V and a maximum output of 30A. It also has two 5V USB ports with a maximum of 2.1A.
- A 12V 18Ah battery.
- An embedded device capable of running the inference model; e.g. a raspberry pi or, in our case, a Jetson Nano.
- An IMX477 Jetson Nano compatible camera.
- A surface for the insects to land on; usually a white sheet [37] and this is what we chose for our trap.
- Multiple LEDs to attract insects into the trap; more about those in subsection 4.3.1.
- A second set of LEDs, to obtain a more natural lighting for our pictures, in order to facilitate our model's recognition process.
- An airtight box to protects components from ambient humidity.
- A way to cool the box; we opted for an aluminium plate screwed to the Jetson Nano heat sink.

- A step down adjustable module DSN mini 360 set to DC/DC 12V input - 5V 1A output to power the LEDs.
- If the 2.1A delivered by the USB ports are not enough to power the embedded device, a 12V output and an LM2596 adjustable DC/DC power converter (1.25V - 35V/3A) can be used to connect the device directly to the charge controller output.

Appendix C

Attributions

C.1 GBIF datasets

The datasets have been made available to us thanks to GBIF and their partners, on the only condition that we cite them using a DOI. So we gladly comply and here are the DOIs of every dataset we used:

- **Hymenoptera:** GBIF.org (11 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.pck2h2
- **Trichoptera:** GBIF.org (11 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.46vce9
- **Orthoptera:** GBIF.org (11 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.x4rjt4
- **Hempitera:** GBIF.org (11 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.9mzfu7
- **Odonata:** GBIF.org (11 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.94qkxy
- **Coleoptera:** GBIF.org (02 April 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.ucgjq6
- **Noctuidae:** GBIF.org (16 March 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.shh46f

- **Geometridae:** GBIF.org (26 March 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.vjyvf2
- **Diptera:** GBIF.org (30 March 2021) GBIF Occurrence Download https://doi.org/10.15468/dl.nsyxvv

If these datasets are not available anymore, you can contact the authors or GBIF at comms@gbif.org.

C.2 Flaticon

Multiple images from flaticon.com were used in this thesis, so we'd like to thank the multiple authors.

Figure 4.3 "Video Camera" by iranshastry on flaticon.com licensed under Flaticon License.

Figure 4.3 "Solar Panel" by monkik on flaticon.com licensed under Flaticon License.

Figure 4.3 "Battery", "Python file symbol" by Freepik on flaticon.com licensed under Flaticon License.

Figure 4.3 "Cpu" by Dimitry Mirollubov on flaticon.com licensed under Flaticon License.

Figure 4.3 "Led" by Smashicons on flaticon.com licensed under Flaticon License.

Figure 4.3 "Charge Controller" by Maurizio Fusillo on flaticon.com licensed under Creative Commons.

Figure 3.1 "Engineering", "Neural" by Becris on flaticon.com licensed under Flaticon License.

Figure 3.1 "Loupe" by Freeplk on flaticon.com licensed under Flaticon License.

Figure 3.1 "Servers", "Tech", "Image gallery", "Nvidia" by Pixel Perfect on flaticon.com licensed under Flaticon License.

C.3 The Object-Detection-Metrics repository

In order to evaluate our algorithms, we used the *Object-Detection-Metrics* repository[42].

Bibliography

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: *arXiv:2004.10934 [cs, eess]* (Apr. 2020). arXiv: 2004.10934. URL: http://arxiv.org/abs/2004.10934 (visited on 05/18/2021).

[2] Cristian Perez Brokate. *Object detection using a Raspberry Pi with Yolo and SSD Mobilenet*. en. Mar. 2019. URL: https://cristianpb.github.io/blog/ssd-yolo (visited on 05/20/2021).

[3] Gilles Charlier. *charliergi/annotate-to-KITTI*. Apr. 2021. URL: https://github.com/charliergi/annotate-to-KITTI (visited on 05/11/2021).

[4] Jian-Wen Chen et al. "A Smartphone-Based Application for Scale Pest Detection Using Multiple-Object Detection Methods". en. In: *Electronics* 10.4 (Jan. 2021). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 372. DOI: 10.3390/electronics10040372. URL: https://www.mdpi.com/2079-9292/10/4/372 (visited on 06/09/2021).

[5] Chris. *What are Max Pooling, Average Pooling, Global Max Pooling and Global Average Pooling?* en-US. Jan. 2020. URL: https://www.machinecurve.com/index.php/2020/01/30/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling/ (visited on 11/07/2020).

[6] Adam Dolezal and Page Baluch. *True Bugs*. en. Text. Oct. 2010. URL: https://askabiologist.asu.edu/explore/true-bugs (visited on 04/15/2021).

[7] *EU Pollinators Initiative - Environment - European Commission*. URL: https://ec.europa.eu/environment/nature/conservation/species/pollinators/policy_en.htm (visited on 04/04/2021).

[8] Pedro F Felzenszwalb et al. *Object Detection with Discriminatively Trained Part Based Models*.

[9] Uwe Fritz. "Order Testudines Batsch, 1788. In: Zhang, Z.-Q. (Ed.) Animal biodiversity: An outline of higher-level classification and survey of taxonomic richness". en. In: *Zootaxa* 3148.1 (Dec. 2011), p. 217. ISSN: 1175-5334, 1175-5326. DOI: 10.11646/zootaxa.3148.1.10. URL: https://biotaxa.org/Zootaxa/article/view/zootaxa.3148.1.10 (visited on 04/15/2021).

[10] Ross Girshick. "Fast R-CNN". In: *arXiv:1504.08083 [cs]* (Sept. 2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083 (visited on 12/02/2020).

[11] Ross Girshick et al. "Region-Based Convolutional Networks for Accurate Object Detection and Segmentation". en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (Jan. 2016), pp. 142–158. ISSN: 0162-8828, 2160-9292. DOI: 10.1109/TPAMI.2015.2437384. URL: http://ieeexplore.ieee.org/document/7112511/ (visited on 11/28/2020).

[12] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". en. In: *arXiv:1311.2524 [cs]* (Oct. 2014). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524 (visited on 11/28/2020).

[13] Caspar A. Hallmann et al. "Declining abundance of beetles, moths and caddisflies in the Netherlands". en. In: *Insect Conservation and Diversity* 13.2 (2020). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/icad.12377, pp. 127–139. ISSN: 1752-4598. DOI: https://doi.org/10.1111/icad.12377. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/icad.12377 (visited on 04/04/2021).

[14] Caspar A. Hallmann et al. "More than 75 percent decline over 27 years in total flying insect biomass in protected areas". en. In: *PLOS ONE* 12.10 (Oct. 2017). Publisher: Public Library of Science, e0185809. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0185809. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0185809 (visited on 04/28/2021).

[15] Muneb ul Hassan. *VGG16 - Convolutional Network for Classification and Detection*. en-US. Section: Popular networks. Nov. 2018. URL: https://neurohive.io/en/popular-networks/vgg16/ (visited on 11/11/2020).

[16] Kaiming He et al. "Mask R-CNN". en. In: *arXiv:1703.06870 [cs]* (Jan. 2018). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870 (visited on 11/30/2020).

[17] Graham Hopkins and Robert Freckleton. "Declines in the numbers of amateur and professional taxonomists: Implications for conservation". In: *Animal Conservation* 5 (Aug. 2002), pp. 245–249. DOI: 10.1017/S1367943002002299.

[18] *How many insect orders are there?* en. URL: https://www.researchgate.net/post/How_many_insect_orders_are_there (visited on 04/21/2021).

[19] *IRM - Avril*. fr. URL: https://www.meteo.be/fr/climat/climat-de-la-belgique/bilans-climatologiques/2021/avril (visited on 05/22/2021).

[20] Amirul Islam, Sen Jia, and Neil D B Bruce. "How much position information do convolutional neural networks encode ?" en. In: (2020), p. 11.

[21] E Yousef Kalafi, C Town, and S Kaur Dhillon. "How automated image analysis techniques help scientists in species identification and classification?" en. In: *Folia Morphol.* 77.2 (2018), p. 15.

[22] Vasudev Kammar et al. "Light Trap: A Dynamic Tool for Data Analysis, Documenting, and Monitoring Insect Populations and Diversity". en. In: *Innovative Pest Management Approaches for the 21st Century: Harnessing Automated Unmanned Technologies*. Ed. by Akshay Kumar Chakravarthy. Singapore: Springer, 2020, pp. 137–163. ISBN: 9789811507946. DOI: 10.1007/978-981-15-0794-6_8. URL: https://doi.org/10.1007/978-981-15-0794-6_8 (visited on 12/31/2020).

[23] Mohamed Esmail Karar et al. "A new mobile application of agricultural pests recognition using deep learning in cloud computing system". en. In: *Alexandria Engineering Journal* 60.5 (Oct. 2021), pp. 4423–4432. ISSN: 11100168. DOI: 10.1016/j.aej.2021.03.009. URL: https://linkinghub.elsevier.com/retrieve/pii/S1110016821001642 (visited on 06/08/2021).

[24] Salman Khan et al. "A Guide to Convolutional Neural Networks for Computer Vision". In: *Synthesis Lectures on Computer Vision* 8.1 (Feb. 2018). Publisher: Morgan & Claypool Publishers, pp. 1–207. ISSN: 2153-1056. DOI: 10.2200/S00822ED1V01Y201712COV015. URL: https://www.morganclaypool.com/doi/10.2200/S00822ED1V01Y201712COV015 (visited on 11/10/2020).

[25] Roel van Klink et al. "Meta-analysis reveals declines in terrestrial but increases in freshwater insect abundances". en. In: *Science* 368.6489 (Apr. 2020). Publisher: American Association for the Advancement of Science Section: Report, pp. 417–420. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aax9931. URL: https://science.sciencemag.org/content/368/6489/417 (visited on 04/04/2021).

[26] Sai Prajwal Kotamraju. *SaiPrajwal95/annotate-to-KITTI*. May 2021. URL: https://github.com/SaiPrajwal95/annotate-to-KITTI (visited on 05/26/2021).

[27] Jason Kuan. *jason9075/opencv-mosaic-data-aug*. Apr. 2021. URL: https://github.com/jason9075/opencv-mosaic-data-aug (visited on 05/19/2021).

[28] Stanford Vision Lab. *ImageNet Large Scale Visual Recognition Competition 2014 (ILSVRC2014)*. URL: http://www.image-net.org/challenges/LSVRC/2014/results (visited on 11/11/2020).

[29] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. en. In: (1998), p. 46.

[30] Suchang Lim, Seunghyun Kim, and Doyeon Kim. “Performance effect analysis for insect classification using convolutional neural network”. en. In: *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. Penang: IEEE, Nov. 2017, pp. 210–215. ISBN: 978-1-5386-3897-2. DOI: 10.1109/ICCSCE.2017.8284406. URL: http://ieeexplore.ieee.org/document/8284406/ (visited on 11/28/2020).

[31] Nabil MADALI. *Demystifying Region Proposal Network (RPN)*. en. May 2020. URL: https://medium.com/@nabil.madali/demystifying-region-proposal-network-rpn-faa5a8fb8fce (visited on 04/29/2021).

[32] *Major Insect Orders Mini-Book - Teachers (U.S. National Park Service)*. en. URL: https://www.nps.gov/teachers/classrooms/insect-orders.htm (visited on 04/15/2021).

[33] Maxime Martineau et al. “A survey on image-based insect classification”. en. In: *Pattern Recognition* 65 (May 2017), pp. 273–284. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2016.12.020. URL: http://www.sciencedirect.com/science/article/pii/S0031320316304411 (visited on 11/10/2020).

[34] Maxime Martineau et al. “Effective Training of Convolutional Neural Networks for Insect Image Recognition”. en. In: *Advanced Concepts for Intelligent Vision Systems*. Ed. by Jacques Blanc-Talon et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 426–437. ISBN: 978-3-030-01449-0. DOI: 10.1007/978-3-030-01449-0_36.

[35] *Mask R-CNN - ShortScience.org*. URL: http://www.shortscience.org/paper?bibtexKey=journals/corr/HeGDG17 (visited on 12/01/2020).

[36] Sancho McCann. *It’s a bird... it’s a plane... it... depends on your classifier’s threshold*. en. Sept. 2011. URL: https://sanchom.wordpress.com/2011/09/01/precision-recall/ (visited on 05/17/2021).

[37] Graham A. Montgomery et al. “Standards and Best Practices for Monitoring and Benchmarking Insects”. English. In: *Frontiers in Ecology and Evolution* 8 (2021). Publisher: Frontiers. ISSN: 2296-701X. DOI: 10.3389/fevo.2020.579193. URL: https://www.frontiersin.org/articles/10.3389/fevo.2020.579193/full?utm_source=S-TWT&utm_medium=SNET&utm_campaign=ECO_FEVO_XXXXXXXX_auto-dlvrit (visited on 04/17/2021).

[38] The Australian Museum. *Ants, Wasps, Bees and Sawflies: Order Hymenoptera*. en. URL: australian.museum/learn/animals/insects/sawflies-wasps-bees-ants-hymenoptera/ (visited on 04/16/2021).

[39] Ard Nieuwenhuizen, Jochen Hemming, and Hyun Suh. “Detection and classification of insects on stick-traps in a tomato crop using Faster R-CNN”. en. In: (), p. 5.

[40] Rafael Padilla. *rafaelpadilla/Object-Detection-Metrics*. May 2021. URL: https://github.com/rafaelpadilla/Object-Detection-Metrics (visited on 05/16/2021).

[41] Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. “A Survey on Performance Metrics for Object-Detection Algorithms”. en. In: *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. Niterói, Brazil: IEEE, July 2020, pp. 237–242. ISBN: 978-1-72817-539-3. DOI: 10.1109/IWSSIP48289.2020.9145130. URL: https://ieeexplore.ieee.org/document/9145130/ (visited on 05/16/2021).

[42] Rafael Padilla et al. “A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit”. In: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030279. URL: https://www.mdpi.com/2079-9292/10/3/279.

[43] *Power Consumption Benchmarks | Raspberry Pi Dramble*. URL: https://www.pidramble.com/wiki/benchmarks/power-consumption (visited on 05/20/2021).

[44] Prabhu. *Understanding of Convolutional Neural Network (CNN) — Deep Learning*. en. Nov. 2019. URL: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148 (visited on 11/07/2020).

[45] Benjamin Price and Ed Baker. “NightLife: A cheap, robust, LED based light trap for collecting aquatic insects in remote areas”. en. In: *Biodiversity Data Journal* 4 (Mar. 2016). Publisher: Pensoft Publishers, e7648. ISSN: 1314-2828. DOI: 10.3897/BDJ.4.e7648. URL: https://bdj.pensoft.net/article/7648/ (visited on 12/31/2020).

[46] *python - Starmap combined with tqdm?* URL: https://stackoverflow.com/questions/57354700/starmap-combined-with-tqdm (visited on 05/02/2021).

[47] Balakrishnan Ramalingam et al. “Remote Insects Trap Monitoring System Using Deep Learning Framework and IoT”. en. In: *Sensors* 20.18 (Jan. 2020). Number: 18 Publisher: Multidisciplinary Digital Publishing Institute, p. 5280. DOI: 10.3390/s20185280. URL: https://www.mdpi.com/1424-8220/20/18/5280 (visited on 12/02/2020).

[48] Joseph Redmon and Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *arXiv:1612.08242 [cs]* (Dec. 2016). arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242 (visited on 05/18/2021).

[49] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv:1804.02767 [cs]* (Apr. 2018). arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767 (visited on 05/18/2021).

[50] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *arXiv:1506.02640 [cs]* (May 2016). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640 (visited on 05/18/2021).

[51] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *arXiv:1506.01497 [cs]* (Jan. 2016). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497 (visited on 12/02/2020).

[52] Vadim Romanuke. “Appropriate Number and Allocation of RELUS in Convolutional Neural Networks”. en. In: *Research Bulletin of the National Technical University of Ukraine "Kyiv Politechnic Institute"* 0.1 (Mar. 2017), pp. 69–78. ISSN: 2519-8890, 1810-0546. DOI: 10.20535/1810-0546.2017.1.88156. URL: http://bulletin.kpi.ua/article/view/88156 (visited on 11/07/2020).

[53] Dan Jeric Arcega Rustia et al. *A real-time multi-class insect pest identification method using cascaded convolutional neural networks*. en.

[54] Francisco Sánchez-Bayo and Kris A. G. Wyckhuys. “Worldwide decline of the entomofauna: A review of its drivers”. en. In: *Biological Conservation* 232 (Apr. 2019), pp. 8–27. ISSN: 0006-3207. DOI: 10.1016/j.biocon.2019.01.020. URL: https://www.sciencedirect.com/science/article/pii/S0006320718313636 (visited on 03/17/2021).

[55] Saleh Shahinfar, Paul Meek, and Greg Falzon. ““How many images do I need?” Understanding how sample size per class affects deep learning model performance metrics for balanced designs in autonomous wildlife monitoring”. en. In: *Ecological Informatics* 57 (May 2020), p. 101085. ISSN: 1574-9541. DOI: 10.1016/j.ecoinf.2020.101085. URL: https://www.sciencedirect.com/science/article/pii/S1574954120300352 (visited on 05/23/2021).

[56] Benno I. Simmons et al. “Worldwide insect declines: An important message, but interpret with caution”. In: *Ecology and Evolution* 9.7 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ece3.5153, pp. 3678–3680. ISSN: 2045-7758. DOI: https://doi.org/10.1002/ece3.5153. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/ece3.5153 (visited on 03/17/2021).

[57] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv:1409.1556 [cs]* (Apr. 2015). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556 (visited on 11/11/2020).

[58] Shrey Srivastava et al. “Comparative analysis of deep learning image detection algorithms”. In: *Journal of Big Data* 8.1 (May 2021), p. 66. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00434-w. URL: https://doi.org/10.1186/s40537-021-00434-w (visited on 05/18/2021).

[59] Home surveillance et al. *Basic motion detection and tracking with Python and OpenCV*. en-US. May 2015. URL: https://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and-tracking-with-python-and-opencv/ (visited on 04/16/2021).

[60] Hironori Takimoto et al. “Using a two-stage convolutional neural network to rapidly identify tiny herbivorous beetles in the field”. en. In: (May 2021). DOI: 10.1101/2021.05.27.445368. URL: http://biorxiv.org/lookup/doi/10.1101/2021.05.27.445368 (visited on 06/09/2021).

[61] *The PASCAL Visual Object Classes Homepage*. URL: http://host.robots.ox.ac.uk/pascal/VOC/ (visited on 06/09/2021).

[62] *Training with mixed precision*. en-us. concept. Archive Location: Training. URL: http://docs.nvidia.com/deeplearning/frameworks/mixed-precision-training/index.html (visited on 05/05/2021).

[63] Udeme Udofia. *Basic Overview of Convolutional Neural Network (CNN)*. en. Sept. 2019. URL: https://medium.com/dataseries/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17 (visited on 11/07/2020).

[64] Furman University and Twitter Twitter. *Does a Cold Winter Decrease Bugs?* en. Section: Treehugger. URL: https://www.treehugger.com/does-a-cold-winter-decrease-bugs-4863437 (visited on 05/22/2021).

[65] Miroslav Valan et al. “Automated Taxonomic Identification of Insects with Expert-Level Accuracy Using Effective Feature Transfer from Convolutional Networks”. en. In: *Systematic Biology* 68.6 (Nov. 2019). Ed. by Thomas Buckley, pp. 876–895. ISSN: 1063-5157, 1076-836X. DOI: 10.1093/sysbio/syz014. URL: https://academic.oup.com/sysbio/article/68/6/876/5368535 (visited on 11/08/2020).

[66] *VGG16 - Convolutional Network for Classification and Detection*. en-US. Nov. 2018. URL: https://neurohive.io/en/popular-networks/vgg16/ (visited on 04/25/2021).

[67] Dujin Wang et al. “Using an Improved YOLOv4 Deep Learning Network for Accurate Detection of Whitefly and Thrips on Sticky Trap Images”. en-us. In: (). URL: https://doi.org/10.13031/trans.14394 (visited on 06/09/2021).

[68] Xiqi Wang et al. “R-YOLO: A Real-Time Text Detector for Natural Scenes with Arbitrary Rotation”. In: *Sensors* 21.3 (Jan. 28, 2021), p. 888. ISSN: 1424-8220. DOI: 10.3390/s21030888. URL: https://www.mdpi.com/1424-8220/21/3/888 (visited on 06/06/2021).

[69] Pete Warden. *How many images do you need to train a neural network?* en. Dec. 2017. URL: https://petewarden.com/2017/12/14/how-many-images-do-you-need-to-train-a-neural-network/ (visited on 05/23/2021).

[70] *Welcome to the DeepStream Documentation — DeepStream 5.1 Release documentation.* URL: https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_Overview.html#nvidia-deepstream-overview (visited on 05/09/2021).

[71] *What, Why and Which?? Activation Functions | by Snehal Gharat | Medium.* URL: https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441 (visited on 04/23/2021).

[72] Denan Xia et al. “Insect Detection and Classification Based on an Improved Convolutional Neural Network”. en. In: *Sensors* 18.12 (Nov. 2018), p. 4169. ISSN: 1424-8220. DOI: 10.3390/s18124169. URL: http://www.mdpi.com/1424-8220/18/12/4169 (visited on 06/08/2021).

[73] H. Yang et al. “Research on insect identification based on pattern recognition technology”. In: *2010 Sixth International Conference on Natural Computation.* Vol. 2. ISSN: 2157-9563. Aug. 2010, pp. 545–548. DOI: 10.1109/ICNC.2010.5583156.

[74] Ryan Zemel and David Houghton. “The Ability of Specific-wavelength LED Lights to Attract Night-flying Insects”. In: *The Great Lakes Entomologist* 50.2 (Feb. 2018). ISSN: 0090-0222. URL: https://scholar.valpo.edu/tgle/vol50/iss2/8.

The sharp decline in insect population during the last decades has highlighted the need for more insect monitoring. Until recently, acquiring data on insect population was done manually and required a significant amount of human workload. To lessen this load, new methods of data collection have been developed using computer vision. Automated systems can recognise insects with the help of AI and monitor the fluctuation of biodiversity in a location. These systems usually consist of a trap, or a drone, to take pictures of the insects. Those pictures are then sent to a central server that performs the recognition.

In our thesis, we chose a different approach by performing every aspect of monitoring directly in our trap, i.e. attracting, taking pictures of and recognising insects. With this difference, we had the possibility to make a completely autonomous trap, as it will work offline. We propose a real-time autonomous monitoring trap that uses YOLOv4 as its object detection model with a ResNet18 backbone. The model has been trained and tested on more than 48000 images taken from GBIF. Our experimental results show that detection on the edge is possible in real-time and with an mAP of 85.33% with ResNet18.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**